Technical University Berlin

Telecommunication Networks Group

Implementation Design of the MOMBASA Software Environment

L. Westerhoff, A. Festag
{westerhoff|festag}@ee.tu-berlin.de

Berlin, November 2001
Version 1.0

TKN Technical Report TKN-01-017

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

In this report the implementation of the *MOMBASA Software Environment* is presented. The *MOMBASA Software Environment* is an experimental platform to examine multicast-based host mobility in IP networks. The report describes the implementation design of the main components (Mobile Agent, MEP[1] and Gateway), the implementation environment and the necessary modifications of the Linux kernel. Main purpose of the report is to document the current implementation of the *MOMBASA Software Environment* and particularly, to facilitate its extensibility for use as a generic toolkit for experimentation with multicast-based mobility support.

[1]Mobility Enabling Proxy

Contents

List of Figures

Chapter 1

Introduction

The *MOMBASA Software Environment* is an experimental platform to investigate multicast-based mobility support in IP networks. The implementation is based on the specification described in [7]. The implementation is publically available at `http://www-tkn.ee.tu-berlin.de/research/mombasa/mse.html` [15].
In the *MOMBASA SE* the following functionalities have been implemented:

- Addressing and routing based on IP- and IP-style multicast.
- Multicast proxies in access points to disburden the mobile host from multicast group management.
- Advertisements/Solicitations to advertise the availability of Mobility-Enabling Proxies (MEPs).
- Inter-MEP advertisements to register mobile hosts in advance.
- Support of different handover types: soft, predictive, inter-technology (vertical) handover.
- Differentiation between active and inactive mobile hosts, multicast-based paging to locate inactive mobile hosts.
- Buffering of data packets for predictive handover and paging.
- Policies to control system behavior (handover type, selection of optimal interface among several possible, control buffering, forwarding algorithm and paging algorithm).

Moreover, the implementation has the following features:

- The implementation supports IP (Version 4).
- It uses IP multicast to support hard, soft and predictive handover. It neither requires any modification to the multicast routing protocol nor does it depend on a certain one.
- It supports heterogeneous networks, namely all technologies which support and are supported by IP. It has support for multiple network interfaces simultaneously in the mobile (potentially of different technologies), and allows handover between access points on different interfaces and therefore handover between different technologies.

- No modification to the correspondent (i.e. fixed) host. That means any IP capable host can communicate with the mobile host. Only the mobile host and the access network must have special preparations for it.
- The system is soft state, i.e. every state eventually times out. This makes the system robust against the breakdown of network links and the crash of components within the system.
- Important behavior, such as selecting an access point for handover or buffering packets during handover, can be tuned by means of policies. This makes the *MOMBASA Software Environment* usable for experimental evaluation of different handover mechanisms and buffer strategies.
- The *MOMBASA SE* can be extended easily to support other multicast schemes.

The following (partly experimental) technologies were used to implement the described environment:

- **Link-layer sockets** were used to intercept packets that should be buffered. Although the *MOMBASA SE* is dealing with IP only, link-layer (i.e. packet) sockets had to be used, since Linux Raw IP sockets are write-only. However, **raw IP sockets** were used for sending buffered data.
- Linux **socket filters** (compatible to Berkeley Packet Filters) were used for the following purposes:
  - In the MEPs each mobile-associated buffer has its own socket. A filter is attached to this socket accepting only packets for the respective mobile host.
  - In the Mobile Agent the socket for idle detection has a quite complex filter that discards signaling (including management interface), multicast and broadcast packets. Thus, any packet that can be read from the socket is a data packet and should reset the activity timer or trigger the wakeup procedure.
- **Network Address Translation** between the unicast and the multicast realm was used. Therefor the NAT code in the Linux kernel was modified.
- The **Ethertap** device, a software device emulating an Ethernet network interface, was used for sending of multicast packets by the gateway.
- Paging was implemented efficiently. Multicast routing support in the Linux kernel was modified to support paging.
- **Multithreading** was used in the mobile host (main and idle thread) and in the MEP (main thread and a thread for each mobile buffer).

The main components of the *MOMBASA Software Environment* are implemented as daemons running in user space with root privileges. Additionally, some modifications to the Linux kernel were necessary. The following components has been implemented:

Mobile Agent The Mobile Agent is responsible for last-hop-signaling (detection of MEPs, registration, handover) and idle detection on the mobile host.

Mobility Enabling Proxy The Mobility Enabling Proxy (MEP) resides on the access point and is responsible for last-hop-signaling (advertisements, handling of registrations), administration of registered mobile hosts, inter-MEP-signaling (advertisement of registered mobiles to pre-register them at neighboring MEPs) and MEP-GWP-signaling (sending of *Paging Updates*, handling of *Paging Requests*).

Gateway Proxy The Gateway Proxy maintains a paging table and controls paging of the mobile host.

NAT from/to multicast Packets must be translated from unicast to multicast in the gateway and back to unicast in the access points. However, the original NAT code in the Linux kernel did not support translation between unicast and multicast realm. Thus, a patch had to be developed. Details are given in Sec. 4.2.

Kernel paging support Paging in the gateway needs some support that can best be implemented in the kernel. Sec. 4.3 gives details on this kernel patch.

This report is structured as follows: In the next section (Sect. 2) the implementation environment is described. In section 3 the design of the agents and in section 4 the necessary modifications if the Linux kernel are presented. Finally, the extensibility of the *MOMBASA SE* is discussed.

Chapter 2

Implementation Environment

The *MOMBASA SE* is implemented for a Linux system running on an x86 architecture (e.g. Intel Pentium, Pentium II, AMD Athlon). It would probably also run on other Linux architectures, however, this was not tested at all. The kernel patches require a standard Linux kernel version 2.2.18. Other versions of the 2.2 series may or may not work.

The *MOMBASA SE* uses only standard C and C++ libraries (including the Standard Template Library). glibc 2.1.3 (and probably higher) is preferred but 2.1.1 will also work if a special switch (`COMPAT`) is set on compilation that enables the definition of some data structures that are missing in the include files of this library version.

The *MOMBASA SE* was developed with a Linux installation based on the S.u.S.E. Linux distribution, however, since only standard libraries and tools (GNU tools) are used, it should work with any Linux distribution providing the necessary kernel and library versions.

Multicast support is provided by the standard Linux kernel (IGMPv2 and kernel support for a PIM-SM Version 2 daemon) and a free multicast routing daemon for PIM-SM (pimd-2.1.0-alpha28) by the University of Southern California's Computer Networks and Distributed Systems Research Laboratory (see [12]). Please note that the implementation of the *MOMBASA SE* does not depend on a specific multicast routing daemon.

Chapter 3

Design of Agents

3.1 Common Design

The three agents Mobile Agent, Mobility Enabling Proxy and Gateway Proxy have the same design. However there is no inheritance relation between the agents. Only the design is reused not the implementation. Inheritance will make only sense if either different classes have to be used in place of each other and thus must have a common interface (this is called polymorphy) or if code shall be reused. The first is not the case with *MOMBASA SE*. The agent classes are in different applications on different hosts and thus need not be polymorphic. The second is only the case on a superficial look at the implementations. On a closer look, most parts are only similar. To make them reusable for all agents would have been the same work as implementing them for each agent.

For this reasons I decided to reuse only the design of the agents and not the implementation. More about re-usage of design can be read in [8].

The *MOMBASA SE* employs mainly two design patterns or design concepts:

All agents are implemented as singletons, i.e. only one instance of the agent class may exist per application. This is ensured by making the constructor protected, thus no instances can be constructed from outside the scope of the agent. A protected static member variable holds a reference to the only agent instance. The instance can be retrieved by a public static method.

The agents employ an event driven concept. Two types of events exist: external and internal events. External events are the reception of protocol messages. Internal events are the expiration of timers.

The design that is common to all agents can be seen in Fig. 3.1[1]. Members or classes that are set in italics are placeholders for members and classes in the agents that have similar purposes but differ in some details.

3.1.1 Data Fields

The following fields can be identified in the agents (field that are only specific to one agent are omitted here and discussed later):

[1]The class diagrams in this chapter are UML class diagrams.

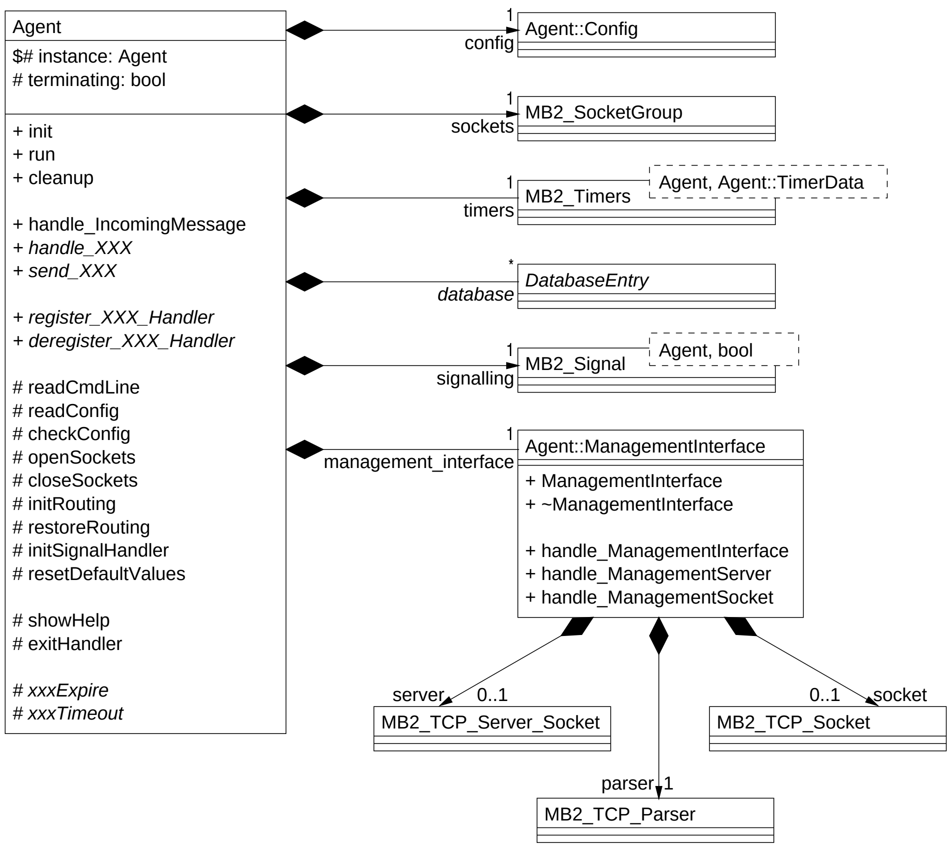


Figure 3.1: Class diagram: Design of agents

instance	Static member holding the only instance of the agent class.
terminating	Flag set by a signal handler[2] or by critical errors to trigger a clean termination at the next possible time.
config	Structure holding configuration options from the configuration file and the command line. Holds also information about configured interfaces and possibly sockets bound to this interfaces.
sockets	Group of sockets that can generate external events. Note that a socket group is not used as a container for sockets but only to group them for the purpose of being watched. See Sec. 3.5.2 for details on how a socket group works.
timers	Chronologically ordered list of timers. A timer encapsulates its expiration time with a callback method that is called on expiration and data passed to this method. See Sec. 3.5.3 for details.

[2]The term signal is used in the sense of a kind of software interrupts in UNIX-like operating systems here.

database	Each agent holds one or more databases. The Mobile Agent stores the known access points, the MEPs store registered mobiles and access points in the same MEP group and the Gateway Proxy stores the pageable mobiles in a hash map that maps unicast IP addresses to the administered objects.
signaling	List that holds the handlers for UNIX signals. It has nothing to do with the protocol signaling.
management_interface	The management interface is a TCP-socket-based interface used for testing. With the management interface handover can be initiated and internal state information can be retrieved.

3.1.2 Methods

The following methods are present in all agents:

Main Methods

init	Does the initialization of the agent (e.g. reading of configuration file, opening of sockets etc.). Calls some helper methods to do so. See below for details.
run	Contains the main event loop. Waits until a message arrives at one of the watched sockets (sockets in the `sockets` group) or a timer expires. This is done by waiting for a socket to become ready to read from and setting the timeout to the expiration time of the next timer. The main message handler (`handle_IncomingMessage`) or the handler of the expired timer is called respectively. If the `terminating` flag is set the event loop will be canceled.
cleanup	Does the cleanup of the agent (closing of sockets etc.). Calls the helper methods described below.

Handling of Messages

handle_IncomingMessage	Main message handler and dispatcher. This method identifies the type of message, reads the message, handles general errors like truncated messages or read errors and dispatches the message to the appropriate handler for this message type.
handle_xxx	Each message type has its own message handler that is called by the main message handler `handle_IncomingMessage`. This handler usually is passed the message together with some context information such as the receiving interface, signal quality, etc. The signal quality is supposed to have a value between 0 and 100 or -1 if not supported. The signal quality can be used as a criterion of access point selection. To implement this a pre or post processing policy handler for the message must be installed that is able to retrieve the signal quality from the MAC layer.

send_xxx	Each message that can be sent by the agent has a corresponding send method. Message fields that can not be filled in from internal fields of the agent are passed as arguments.

Handling of Timers Methods that are called on the expiration of a timer are set on the initialization of the timer. One argument is passed to the timer method. This argument is also set with the timer. This argument usually contains a pointer to an entity the timer is responsible for (e.g. a mobile host entry).

xxxExpire/xxxTimeout/xxxTimer	Timer methods either called in normal operation, usually to maintain soft-state behavior or called in exceptional conditions, e.g. when an expected message did not arrive in time.

3.1.3 Initialization and Cleanup Helper Methods

The following protected methods are called by either `init` or `cleanup`:

readCmdLine	Parses the command line and sets the appropriate flags in `config`. Initialization method that is called by `init`.
readConfig	Parses the config file and sets the appropriate variables in `config`. Initialization method that is called by `init`.
checkConfig	Checks if the configuration is consistent. The validity of single configuration options is already checked in `readConfig`. This method checks only the inter-operation between several configuration options. Initialization method that is called by `init`.
openSockets	Creates and opens sockets that are necessary for the operation of the agent. Initialization method that is called by `init`.
closeSockets	Closes sockets used by the agent. Cleanup method that is called by `cleanup`.
initRouting	Configures routing. Initialization method that is called by `init`.
restoreRouting	Restores routing to the state before start of the agent. Cleanup method that is called by `cleanup`.
initSignalHandler	Initializes the UNIX signal handlers. Initialization method that is called by `init`.
resetDefaultValues	Sets default values for all members. Method that is called by the constructor and by `cleanup`.

3.1.4 Policies

The handling of messages, selection of access point in the Mobile Agent, buffering and flushing buffers can be fine-tuned by policies. A policy is represented by a template class (`MB2_PolicyHandler`, see below) that has methods for pre- and post-processing. The argument and return types of the processing methods are template parameters.

Registration and de-registration methods for each type of policy handler are provided by the agent class.

The policy handlers are stored in ordered lists (implemented as maps). In most cases multiple handlers can be registered for one message. Two phases exist for message processing. Pre-processing handlers are called before the standard message processing, post-processors are called afterwards. The handlers are executed in the order of their preference until one of them returns a non-zero return code. If the return code is negative the messages processing will be aborted.

The only exceptions to this rule are the SelectBS policy in the Mobile Agent and the buffer and flush policies. Of these handlers only one may be registered and they have only one phase. The latter two policy handlers are not represented by the `MB2_PolicyHandler` template but by own classes.

3.1.5 Management Interface

The management interface can be used to externally control handover and to retrieve internal state information. This can be used for testing. The management interface provides a TCP server socket on a configured port. This server accepts one single connection at a time. Text commands from this connection are interpreted by a parser and replies are given in a text-based format. Commands should be given to the interface in complete lines at a time. The connection should be closed only with a `close` command. Valid commands are different for each agent and are discussed in the corresponding agent specific sections.

3.2 Design of Mobile Agent

The design of the Mobile Agent can be seen in Fig. 3.2. Members that are common to the design of all agents were already presented in the previous section. Access methods and the policy registry are also omitted here.

3.2.1 Data Fields

state	State of the mobile. Valid states are: **`NON_INITIALIZED`** The Mobile Agent has not been initialized, yet. **`INITIALIZED,WAIT4MEP`** The Mobile Agent is initialized and is waiting for a *MEP Advertisement*. **`REG_PENDING`** The Mobile Agent has no valid registration but has sent a registration message to a MEP for which a reply was not received, yet. **`ACTIVE`** The Mobile Agent is active and has a valid registration with a MEP. **`INACTIVE`** The Mobile Agent is inactive and is not waiting for a *MEP Advertisement*.
active	Set when the mobile is active. Needed to distinguish between active and inactive mobiles when in state `WAIT4MEP`.
wakeup_pipe	UNIX pipe used as a communication channel between the main thread and the idle thread. The idle thread writes a dummy value to the pipe on wakeup. The wakeup trigger appears in the main thread like an external message.
waking_up	The Mobile Agent should wake up but a valid registration could not be established, yet.
last_id	Identifier of the last sent *MH Registration Request*.
basestations	Database of known access points hashed by their IP address.
registered_bs	Pointer to the entry of the access point we are registered at (if any). Only set in state `ACTIVE`.
pending_reg	Information about a *MH Registration Request* for which a *MH Registration Reply* is expected, but was not received, yet. May be set in states `ACTIVE` and `PENDING_REG`. The info structure contains the message, a pointer to the related `BaseStationEntry` and the sending timestamp.
reg_retry_count	When a reply to a registration was not received in time, we retry sending of the request. This counts how many retries are left.
regreq_timeout	Timeout, within which a reply to a registration request should be received, expired.
rereg_timer	Timer expires when it is time to re-register at an access point to refresh state information. Usually one third of the registration lifetime.
reg_timer	Timer expires when the lifetime of the current registration expires. Usually this means that re-registrations failed.

default_route_saved	The default route was saved on initialization of the agent.
orig_default_route	Saved default route.
default_route	Current default route.
last_activity	Timestamp (in microseconds) of the last sent or received data packet. Used for the purpose of idle detection.
activity_mutex	Mutex[3] that protects `last_activity` from concurrent access by main and idle thread.
activity_timeout	Timer for checking of an idle condition.
idle_thread	Thread handle of the idle thread.
idle_thread_started	The idle thread was started, i.e. `idle_thread` contains a valid handle.

Configuration

iface	Map of configured interfaces, indexed by the interface name.
paging_enabled	The Mobile Agent is pageable. Also used as a switch for inactive mode support.
broadcast	The mobile host wants to receive broadcast packets. This feature was not implemented due to lack of time.
predictive	The mobile host wants to be pre-registered at neighboring access points.
reg_port	UDP port used for registration messages.
idle_timeout	After how many seconds of no data activity should we go to idle mode?
management_port	TCP port of the management interface (zero if turned off.).
dummy_dev_name	Name of the dummy device.
dummy_dev	`MB2_Interface` object of the dummy device.
foreground	Agent should run in foreground.
config_file	Name of the configuration file.

Interface The `Interface` structure augments the abstraction of network interfaces (as represented by the `MB2_Interface` class) with interface dependant configuration options. Note that the preferred registration lifetime, timeout values and the sending of de-registrations are configured here, as they may depend on the used technology (e.g. values optimal for wired connection via 100Mbps Ethernet may be different than optimal values for GSM).

iface	`MB2_Interface` object of the network interface.
force_addr	IP address used for the interface. If none has been given this will be the first IP address retrieved from the kernel.
preference	Preference of the interface as set in the configuration file. Can be used by the handover policy to select an access point.
active_regtime	Preferred registration lifetime in seconds for active registrations on this interface (can be limited by the maximum lifetime advertised by access points).
idle_regtime	Preferred registration lifetime in seconds for inactive registrations on this interface.

[3]Mutual exclusion device used in multithreded environments. Protects shared data from concurrent modifications.

regreq_timeout	Timeout within which a *MH Registration Reply* to an active registration is expected. Otherwise the *MH Registration Request* is resent.
send_dereg	Set if the agent should try to send de-registration messages to access points at this interface.
adv_socket	ICMP socket for the reception of *MEP Advertisements* and the sending of *MEP Solicitations*. Bound only to this interface.
reg_socket	UDP socket for sending of *MH Registration Requests* and reception of *MH Registration Replies* and *Paging Requests*. Bound only to this interface.

Access Point Database Known access point (called base stations within the implementation) are stored in a hash table that hashes them by their IP address. They are stored in a `BaseStationEntry`.

addr	IP address of the access point. Different interfaces of the access point (possibly with different technologies) have different IP addresses and are thus treated as different access points.
registered	This is the access point we are registered at.
busy	This access point had the busy flag set in its last advertisement and thus does not want to receive new advertisements.
stale	This access point reacted strange previously and thus should be avoided. An access point with this flag will time out even if a refreshing advertisement is received.
quality	Signal quality of the last received advertisement or -1 if not supported.
adv_seqno	Sequence number of the last received advertisement.
adv_lifetime	Lifetime of the last received advertisement.
maxregtime	Maximum lifetime for registrations as advertised.
adv_flags	Flags in the last received advertisement.
timer	Expiry timer for this entry.

3.2.2 Methods

Message Handlers

handle_MEP_Advert	Handler for *MEP Advertisements*. This handler is passed an info structure that contains the raw packet, a pointer to the receiving interface and the signal quality. On entry of the post-processing handlers it contains also a pointer to the corresponding. `BaseStationEntry`. The signal quality must be filled in by a policy handler. The handler updates the access point (i.e. base station) database.
handle_MH_RegReply	Handler for *MH Registration Replies*. The passed info structure contains the message, the receiving interface, signal quality and access point address and port.
handle_PagingReq	Handler for *Paging Requests* that is passed the message and the receiving interface. The handler triggers the wakeup procedure.

Sending of Messages

send_MEP_Solicit	Sends a *MEP Solicitation* on the given interface.
send_MH_RegReq	Sends a registration or de-registration to the given access point.

Mobile-specific Methods

selectBS	Returns the access point that should be used for the next registration. Calls either the corresponding policy handler or returns the first feasible access point.
wakeup	Wakes up from idle mode. Sends an active registration to a suitable access point.
goIdle	Switches to idle mode. Send inactive registration, de-register and cancel pending registration if applicable.

Initialization and Cleanup

startIdleThread	Starts the idle thread. The purpose of the idle thread is described below.
stopIdleThread	Stops the idle thread.

Handling of Timers

baseExpire	Called when an base station entry expires. A pointer to the entry is passed.
regreqTimeout	Called when an expected registration reply did not arrive in time.
regTimeout	Called when the registration at the current access point expires. Usually the registration should be refreshed by the next timer.
reregExpire	Called when it is time to refresh the current registration.
activityTimeout	Called when it is time to check for an idle condition.

Idle Thread

idleThread	Static method that constitutes the main method of the idle thread (see next section).
idleThreadCleanup	Cleanup for the idle thread.
getAddrFromIPH	Function that extracts the destination address from the IP header when flushing buffers.

3.2.3 Idle Detection

One essential property of the mobile host is the distinction between active and inactive mode. The mobile host has to switch from active to inactive mode after a configurable idle period (no IP data traffic). The idle detection works as follows:

A packet socket, which receives all incoming and outgoing packets of the host, is opened. A socket filter, which only accepts IP data packets but rejects non-IP, signaling and broadcast packets, is attached to the socket. In active mode the so-called idle thread does nothing than observing the packet socket and recording the timestamp of the last data packet. The main thread sets a timer to the duration of the activity timeout. When the timer expires, the timestamp of the last data packet is checked. If it lies within the last timer period the timer will be prolongated, otherwise the mobile host will switch to inactive mode. The change from inactive to active mode is triggered by the reception of an incoming or outgoing packet by the idle thread or by a *PAGING_REQUEST*. In the case of outgoing data the packets are buffered by the idle thread until a valid registration at an access point can be obtained. To cause outgoing packets to appear on the idle socket but not be sent out on an interface a dummy device is used, a software network device which discards any packets. In inactive mode the default route points to this device.

3.2.4 Management Interface

The management interface of the Mobile Agent understands the following commands. For the syntax see the paragraph on `MB2_Parser`. Note that the management interface will collide with idle detection if TCP packets are fragmented. This will usually not be the case if client and server are on the same link or if path MTU discovery is performed.

handover	Triggers a handover. The optional argument denotes the address of the access point to handover to.
getState	Prints the current state as symbolic state name, value of the active and wakeup flags (0/1).
getBaseStation	Prints the access point with the given address or a table of all known access points finished by `end` in the following format: IP address, registered flag, busy flag, stale flag (0/1), quality, interface name, sequence number, lifetime, maximum registration time, flags and expiry time in sec,usec.
getRegBaseStation	Prints the access point we are registered at in the same format.
getPendingReg	Prints the pending registration as a comma separated list of the following fields: access point address, flags, lifetime, extended flags, identifier (high,low), sending time (sec,usec).
getRegRetryCount	Get the registration retry counter.
getRegTimer	Get the expiry time of `regtimer` as sec,usec.
getRegReqTimeout	Get the expiry time of `regreqtimeout` as sec,usec.
getReregTimer	Get the expiry time of the `reregtimer` as sec,usec.
getActivityTimeout	Get the expiry time of the `activitytimeout` as sec,usec.
getLastActivity	Get the timestamp of the last activity as sec,usec.
close	Close the connection.
terminate	Shutdown the agent.
reset	Shutdown and restart the agent.

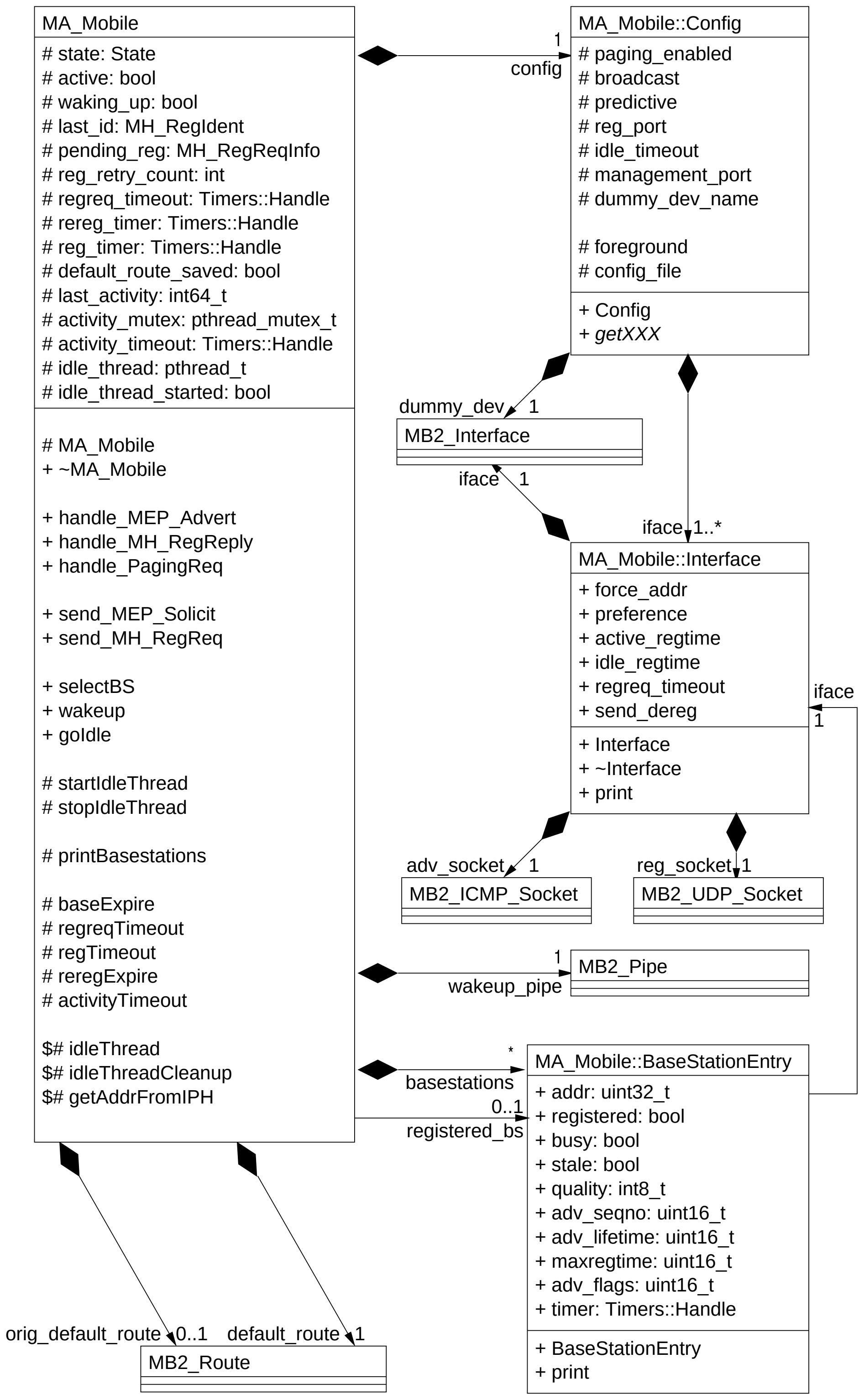


Figure 3.2: Class diagram: Design of the Mobile Agent

3.3 Design of Mobility Enabling Proxy

3.3.1 Data Fields

initialized	The agent is initialized.
direct_mobiles	Number of directly registered mobiles. The total number of mobiles can be retrieved from the mobile database. The number of indirectly registered can be calculated from this.
predictive_mobiles	Number of directly registered mobiles that wish to be pre-registered at other access points. Used for sending of *IMEP Advertisements*
. **nat_route**	Network route that translates multicast addresses to unicast addresses of mobile hosts.
blackhole	Network route to the (sub)network of mobile hosts that prevent packets for mobiles that are not directly registered from being forwarded. Each directly registered mobile has its own host route.
old_netroute_saved	The original route to the mobile network was saved.
old_netroute	The original network route.
mobiles	Database of known mobiles hashed by their unicast IP address. The mobiles are either directly registered with *MH Registration Requests* or pre-registered with *IMEP Advertisements* from other access points.
otherBaseStations	Database of known other access points hashed by their IP address. Collected through *IMEP Advertisements*

Configuration

upstream	Configuration of the upstream interface.
downstream	Map of downstream interfaces, indexed by their names.
send_paging_updates	*Paging Updates* should be sent to the gateway proxy.
paging_proxy	IP address and port of the gateway proxy.
mobile_uc	Unicast address range of the mobile hosts.
mobile_mc	Multicast address range of the mobile hosts.
mobile_mask	Address mask for the address ranges.
mobile_mask_len	Length of the address mask.
tunnel_type	Is the data transported via NAT to/from multicast or via encapsulation in multicast packets. Only NAT is implemented.
reg_port	Registration UDP port number.
imep_port	UDP port for *IMEP Advertisements*.
management_port	TCP port of the management interface (zero if turned off).
paging_area_count	Number of paging areas this MEP is member of.
paging_area	Multicast groups for paging areas. The first is the paging area we report as location for directly registered mobiles to the gateway proxy.
mep_group_count	Number of MEP groups this MEP belongs to including the one, we are only sending to.
mep_group	Multicast groups for MEP groups. The first one is the group centered around us, i.e. the one this MEP is only sending to.

max_direct_mobiles	Maximum allowed number of directly registered mobiles.
max_indirect_mobiles	Maximum allowed number of indirectly registered mobiles.
mobile_threshold	Not used.
max_regtime	Maximum accepted registration time in seconds.
mobile_buffer_size	Size of mobile buffers in bytes.
foreground	Agent should run in foreground.
config_file	Name of the configuration file.

US_Interface This structure augments the interface object for the upstream interface with configuration data.

iface	`MB2_Interface` object of the network interface.
force_addr	IP address that should be used for the interface. If none has been given this will be the first IP address retrieved from the kernel.
imep_interval	Interval between *IMEP Advertisements* this agent sends in seconds.
imep_lifetime	Lifetime of sent *IMEP Advertisements*.
imep_adv_timer	Timer for the regular sending of *IMEP Advertisements*.
imep_seqno	Sequence number of the next *IMEP Advertisements*.
imep_socket	Socket for sending and reception *IMEP Advertisements*.
shared_imep_socket	Do we share the `imep_socket` with a registration socket? Not supported in this version.

Interface The `Interface` structure augments the abstraction of downstream network interfaces (as represented by the `MB2_Interface` class) with interface dependant configuration options. Note that the advertisement interval and lifetime are configured here, as they may depend on the used technology (e.g. values optimal for wired connection via 100Mbps Ethernet may be different than optimal values for GSM).

iface	`MB2_Interface` object of the network interface.
force_addr	IP address that should be used for the interface. If none has been given this will be the first IP address retrieved from the kernel.
adv_interval	Interval between sent *MEP Advertisements* in microseconds.
adv_lifetime	Lifetime for sent *MEP Advertisements* in microseconds.
adv_timer	Timer for the regular sending of *MEP Advertisements*.
adv_seqno	Sequence number of the next advertisement.
adv_socket	Socket for sending of *MEP Advertisements* and reception of *MEP Solicitations*. Bound only to this interface.
reg_socket	Socket for reception of *MH Registration Requests*, sending of replies and *Paging Requests*. Bound only to this interface.

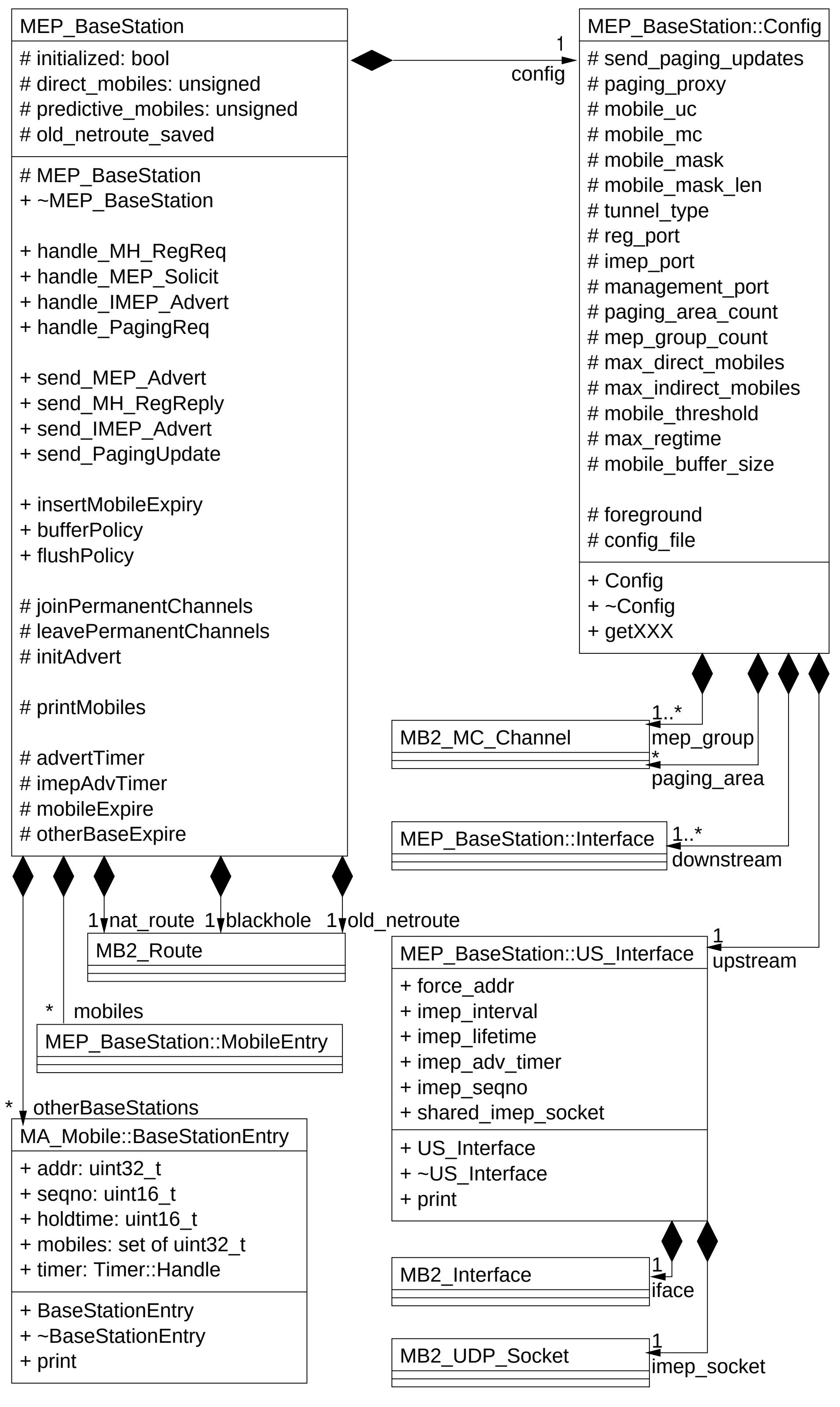


Figure 3.3: Class diagram: Design of the Mobility Enabling Proxy (1)

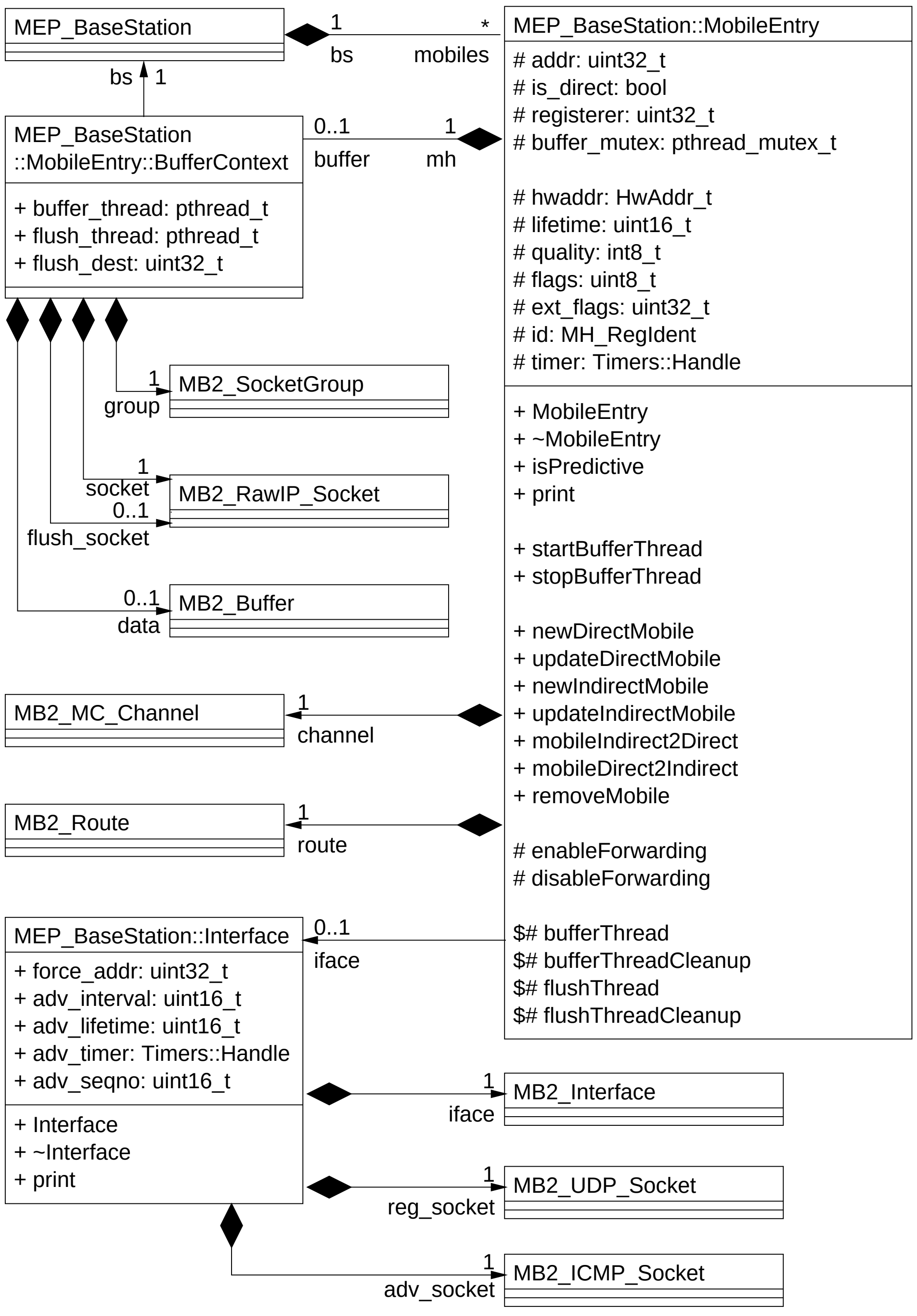


Figure 3.4: Class diagram: Design of the Mobility Enabling Proxy (2)

Mobile Database The mobile database, unlike the access point database, contains active objects (of class `MobileEntry`) not passive data structures. The updating of entries, buffering etc. is done by the entry objects itself.

Data Fields

addr	Unicast IP address of the mobile host.
bs	Back pointer to the enclosing agent.
is_direct	This mobile is directly registered.
registerer	If a valid indirect registration exists this will contain the IP address of the registering MEP.
buffer	Context information (including the buffer itself) for the buffer thread.
buffer_mutex	Mutex for concurrent access to the buffer context.
channel	Multicast group for this mobile.
route	Host route to this mobile host.
hwaddr	Hardware (i.e. MAC) address of the mobile host. Not implemented.
iface	Interface from which a direct registration was received.
lifetime	Lifetime of the direct registration in seconds.
quality	Signal quality of the direct registration (if supported and set by a policy handler).
flags	Flags of the last direct registration.
ext_flags	Extended flags of the last direct registration.
id	Identifier of the last direct registration.
timer	Expiry timer for this entry if the mobile has been directly registered. Otherwise this entry will be controlled by the access point entry of the registering MEP.

Housekeeping Methods

is_predictive	Checks if the mobile is a directly registered one that wishes to be pre-registered at other MEPs.
newDirectMobile	Initializes the entry with data from the *MH Registration Request*, enables forwarding to the last hop and subscribes to the mobile's multicast channel.
updateDirectMobile	Updates the entry with data from the *MH Registration Request*
newIndirectMobile	Initializes the entry for an indirectly registered mobile, starts the buffer thread and joins the multicast group.
updateIndirectMobile	Updates the registerer of this entry.
mobileIndirect2Direct	Turns the indirect mobile into a direct one. Enables forwarding to the last hop, stops buffering and starts flushing.
mobileDirect2Indirect	Turns the direct mobile into an indirect one. Disables forwarding and starts buffering.
removeMobile	Prepares the removal of this mobile entry by disabling forwarding for direct mobiles, stopping buffering for indirect ones and un-subscribing from the multicast channel.

Support Methods

enableForwarding	Enables forwarding to the mobile host by setting a host route to the last hop overriding the network blackhole route.
disableForwarding	Disables forwarding by removing the host route. Thus the blackhole network route for the mobile subnetwork causes the packets to vanish in this host. However the raw sockets feeding the buffers still get the packets.

Buffering When a mobile is registered indirectly by another MEP, packets for the mobile are buffered by this MEP to reduce loss of packets when the mobile does a handover to this MEP. Each mobile entry has its own buffer, its own raw socket and its own buffer thread. The socket is attached a filter only accepting packets for this mobile. The buffer thread runs in an infinite loop and is canceled from the main thread either when the mobile entry becomes invalid or the mobile becomes directly registered. In the latter case a flush thread is started to forward buffered data to the mobile host.

BufferContext

bs	Pointer to the agent instance.
mh	Pointer to the embedding mobile entry.
group	Socket group. Contains only one socket. This is a bug workaround.
socket	Raw IP socket for the data to be buffered.
buffer_thread	Handle for the buffer thread.
data	The ring buffer.
flush_thread	Handle for the flush thread.
flush_dest	Destination for the flushed packets (the mobile's unicast IP address).
flush_socket	Raw IP socket for flushing the buffer to the mobile host.

Buffer related Methods

startBufferThread	Starts buffering of data for the mobile host. The buffer context and the buffer are allocated and a raw socket is opened.
stopBufferThread	Stops the buffering for the mobile host. Either the flushing thread is started (this is the case, when a pre-registered mobile registers directly) or the buffer content is discarded and memory freed (this happens when the inactive registration becomes invalid, because either the entry of the registering MEP expires or the mobile does not appear in the IMEP Advertisements of the registerer anymore).

bufferThread	The buffer thread. This is a static method, however, a buffer thread is created for each indirect mobile. It is passed a `BufferContext` structure including a pointer to the mobile for which to buffer. Since it works only on this structure it can be seen as a logical member of this structure. Runs in an infinite loop unless canceled by the main thread. Reads data for the mobile host from the raw IP socket and writes it to the ring buffer. Whether a packet should be buffered can be controlled by the buffer policy which operates on the IP header.
bufferThreadCleanup	Cleanup method for the buffer thread called on its cancellation.
flushThread	The flush thread. The same facts as for the buffer thread apply. The flush thread translates the buffered packets to unicast, and forwards them to the mobile. This can also be controlled by a policy e.g. to avoid forwarding of aged packets. The flush thread terminates when flushing is finished.
flushThreadCleanup	Cleanup method for the flush thread.

Access Point Database

addr	IP address of the access point.
seqno	Sequence number of the last received IMEP advertisement.
holdtime	Lifetime of the last received IMEP advertisement.
mobiles	Set of the unicast IP addresses of the advertised mobiles in the last IMEP advertisement. Used to check which mobiles to remove and to add on a new advertisement.
timer	Expiry timer for this entry.

3.3.2 Methods

Message Handlers

handle_MH_RegReq	Handler for *MH Registration Requests*. Creates or updates mobile entries (or removes them in the case of de-registrations or inactive registrations), joins the multicast group if appropriate, sends a *Paging Update* for inactive mobiles. The handler is passed an info structure that contains context information about the message, such as the receiving interface, signal quality (if filled in by a policy handler) etc.
handle_MEP_Solicit	Handler for *MEP Solicitations*. Schedule the sending of a *MEP Advertisement* on the receiving interface. The advertisement is not sent instantly to avoid synchronizing effects with other MEPs.
handle_IMEP_Advert	Handler for *Inter-MEP Advertisements*. Updates the access point database, removes old mobiles from the mobile database and inserts new ones. Calculates the difference set of the mobile addresses in the last advertisement and those in the current advertisements to do so.
handle_PagingReq	Handler for *Paging Requests*. Forwards the request to all downstream interfaces.

Sending of Messages

send_MEP_Advert	Sends a *MEP Advertisement* on the given interface.
send_MH_RegReply	Sends a *MH Registration Reply* with a given code that is a reply to the passed request.
send_IMEP_Advert	Sends an *IMEP Advertisement* with the given holdtime (i.e. lifetime).
send_PagingUpdate	Sends a *Paging Update* for the given mobile host to the gateway proxy. The identifier of the triggering *MH Registration Request* and the lifetime of this update are also passed as arguments.

MEP-specific Methods

insertMobileExpiry	Inserts an expiry timer for a mobile entry into the timer list.
bufferPolicy	Calls the buffer policy handler if one exists. The handler is passed the IP header of the packet in question. It controls whether the packet should be buffered. If no policy handler is installed, all packets will be accepted.
flushPolicy	Calls the flush policy handler if one exists. The handler is passed the reception timestamp and the IP header of the packet in question and controls whether the packet should be forwarded to the mobile or discarded. All packet will be forwarded if no policy handler is installed.

Initialization and Cleanup

joinPermanentChannels	Joins the permanent multicast groups such as the paging areas and the MEP groups.
leavePermanentChannels	Leaves the permanent multicast groups.
initAdvert	Schedule the sending of the first *MEP Advertisements* on the downstream interfaces and of the first *IMEP Advertisement*. The sending times are randomized to avoid synchronization effects.

Handling of Timers

advertTimer	It is time to send a *MEP Advertisement* on a certain interface. A new timer is set. Although the advertisements are send frequently, the exact time is randomized to avoid synchronization effects.
imepAdvTimer	It is time to send a new *IMEP Advertisement*. A new timer is set which is also randomized.
mobileExpire	A direct mobile entry expired. The entry will either be removed or turned into an indirect entry if there is an indirect registerer.
otherBaseExpire	The entry of an access point expired. This entry and the mobile entries of all mobiles indirectly registered by this access point are deleted.

3.3.3 Management Interface

The management interface of the Mobility Enabling Proxy understands the following commands. For the syntax see the paragraph on `MB2_Parser`.

getState	Prints whether the agent was initialized (0/1).
getMobile	Prints the entry of the mobile with the given address or a table of all known mobiles finished by `end` in the following format: unicast IP address, direct flag, interface name, lifetime, quality, flags, extended flags, registration id (high,low), expiration time (sec,usec) and IP address of registerer.
getBaseStation	Prints the access point with the given address or a table of all known access points finished by `end` in the following format: IP address, sequence number, holdtime, number of advertised mobiles and expiry time in sec,usec.
getDirectMobiles	Prints the number of directly registered mobiles.
getPredMobiles	Prints the number of directly registered mobiles that want to be pre-registered at other MEPs.
close	Close the connection.
terminate	Shutdown the agent.
reset	Shutdown and restart the agent.

3.4 Design of Gateway Proxy

3.4.1 Data Fields

initialized	The agent is initialized.
paging_socket	Special raw IP socket that forms the connection of the agent to the kernel support and used to control it. The packets that have to be buffered during paging are sent up by the kernel through this socket.
flush_socket	Netlink socket that forms the connection to the *Ethertap* device. Used for flushing of buffers towards the mobile hosts and for sending of *Paging Requests*.
nat_route	Network route that translates unicast addresses to multicast addresses of mobile hosts.
old_netroute_saved	The original route to the mobile network was saved.
old_netroute	The original network route.
mobiles	Database of known mobiles hashed by their unicast IP address. The mobiles are reported by the MEPs with *Paging Updates*.
last_paging_seqno	Sequence number used for the last *Paging Requests*.

Configuration

upstream	Configuration of the upstream interface.
downstream	Map of downstream interfaces, indexed by their names.
mobile_uc	Unicast address range of the mobile hosts.
mobile_mc	Multicast address range of the mobile hosts.
mobile_mask	Address mask for the address ranges.
mobile_mask_len	Length of the address mask.
tunnel_type	Is the data transported via NAT to/from multicast or via encapsulation in multicast packets. Only NAT is implemented.
paging_port	UDP port number used for paging.
management_port	TCP port of the management interface (zero if turned off).
paging_source	IP address used as the source for *Paging Requests*. Must be on the *Ethertap* network.
paging_areas	Multicast address range of paging areas.
paging_areas_mask	IP Mask for paging areas.
paging_areas_masklen	Mask length for paging areas.
max_mobiles	Maximum number of mobile entries.
max_cachetime	Maximum time in seconds that a cache entry may exist without being refreshed (i.e. the maximum lifetime of the paging table entries).
mobile_buffer_size	Size of mobile buffers in bytes.
ethertap_unit	Unit number for the used *Ethertap* device (e.g. 1 for "tap1").
ethertap_name	Name of the Ethertap device (e.g. "tap1").
ethertap_dev	`MB2_Interface` object for the Ethertap device.
foreground	Agent should run in foreground.
config_file	Name of the configuration file.

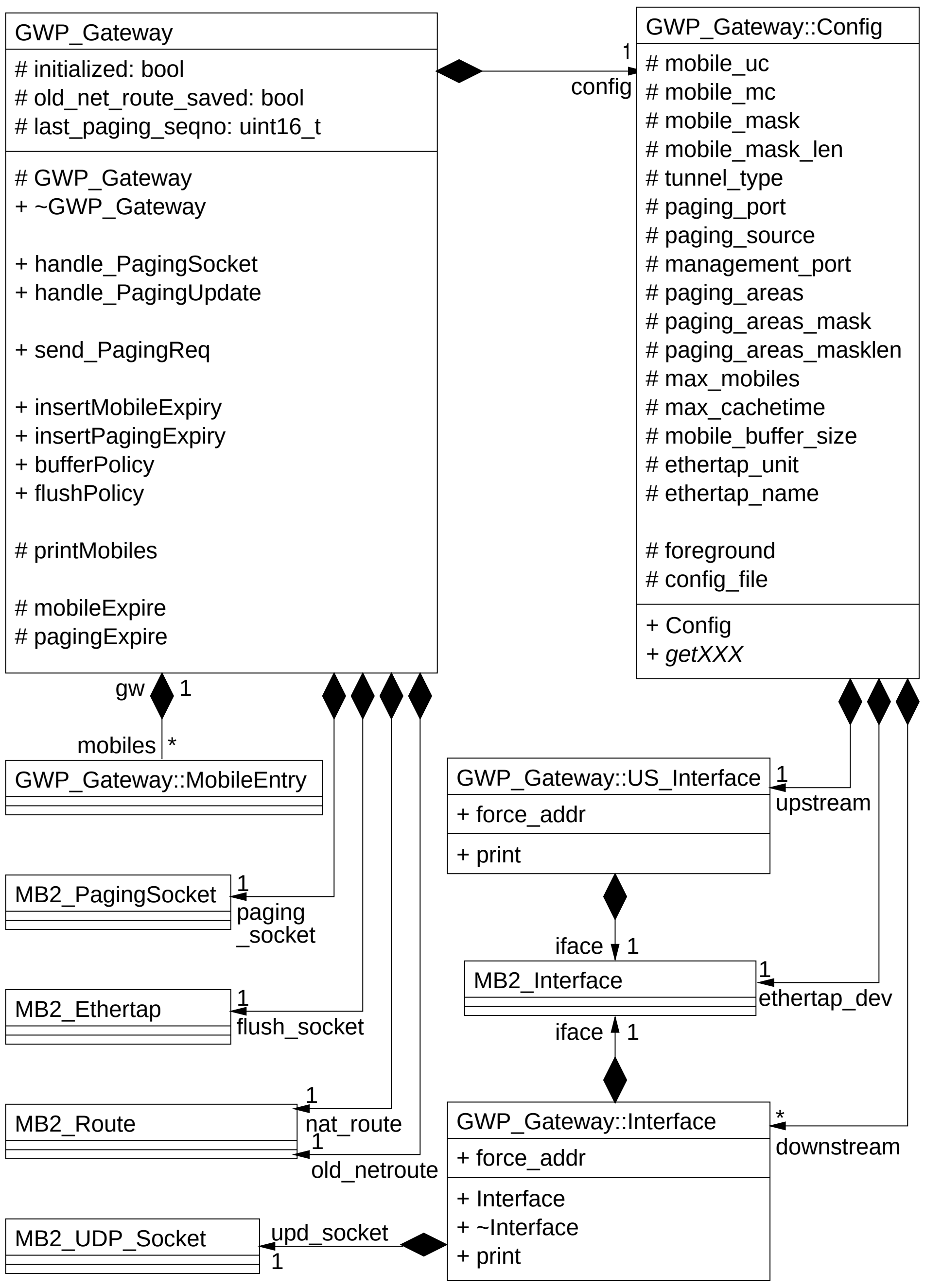


Figure 3.5: Class diagram: Design of Gateway Proxy (1)

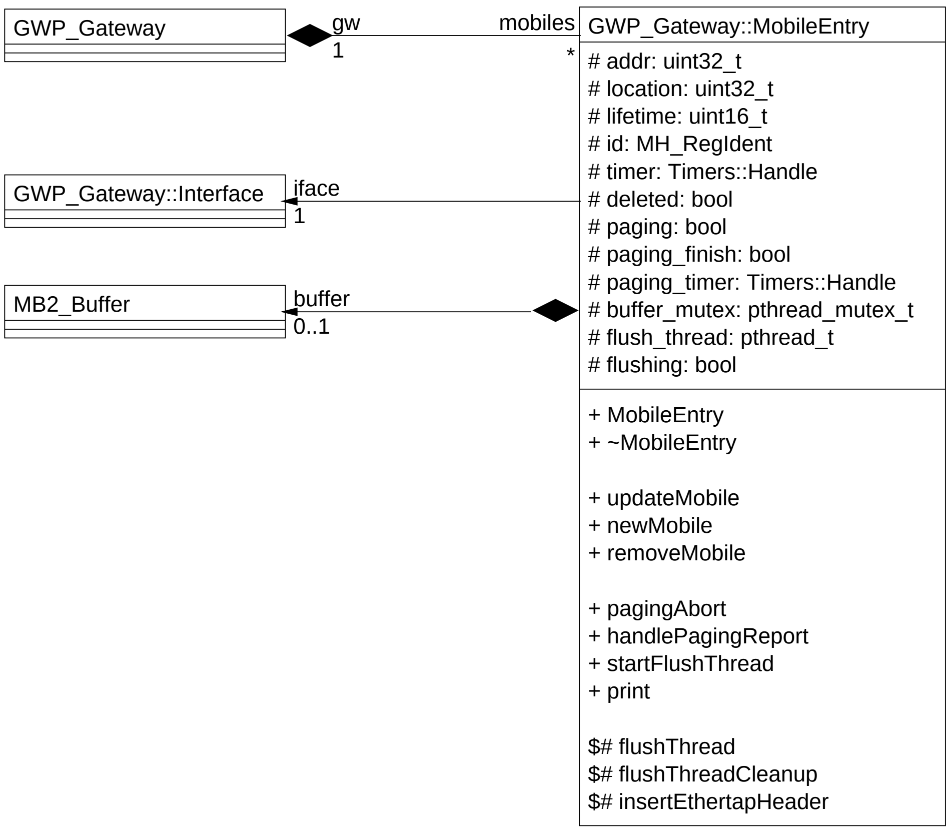


Figure 3.6: Class diagram: Design of Gateway Proxy (2)

US_Interface This structure augments the interface object for the upstream interface with configuration data.

iface	`MB2_Interface` object of the network interface.
force_addr	IP address that should be used for the interface. If none has been given this will be the first IP address retrieved from the kernel.

Interface The `Interface` structure augments the abstraction of downstream network interfaces (as represented by the `MB2_Interface` class) with interface dependant configuration options.

iface	`MB2_Interface` object of the network interface.
force_addr	IP address that should be used for the interface. If none has been given this will be the first IP address retrieved from the kernel.
upd_socket	Socket for the reception of *Paging Updates*. Bound only to this interface.

Mobile Database The mobile database contains active objects not passive data structures. The updating of entries, buffering, handling of paging messages from the kernel etc. is done by the entry objects itself.

Data Fields

gw	Back pointer to the enclosing agent.
addr	Unicast IP address of the mobile host.
location	Multicast address of the paging area reported in the last *Paging Update* as the location of the mobile host.
iface	Interface from which the last *Paging Update* was received.
lifetime	Lifetime of the last *Paging Update*.
id	Registration identifier of the last *Paging Update*. Used to detect reordering.
timer	Expiry timer of this entry.
deleted	The entry is not valid anymore. The entry is not deleted instantly to detect reordering of *Paging Updates*.
paging	Paging for this mobile is just in progress, i.e. a *Paging Request* was sent but a *Paging Updates* with lifetime zero was not received, yet.
paging_finish	We are in the finishing phase of the paging for this mobile, i.e. the *Paging Updates* with lifetime zero was received and we are now waiting for a report by the kernel that the multicast group corresponding to the mobile becomes existent.
paging_timer	Timer that is kept running during the paging process and expires when the kernel does not send a refresh message.
buffer	Buffer for packets for the mobile host that are buffered during the paging phase.
buffer_mutex	Mutex used to synchronize the flush thread with the main thread on destruction of the mobile entry.
flush_thread	Thread handle of the corresponding flush thread.
flushing	Just flushing the buffer. Only if this is true there will be a flush thread.

Housekeeping Methods

newMobile	Initializes the entry with data from the *Paging Update* and potentially deletes the corresponding entry in the kernel Multicast Forwarding Cache.
updateMobile	Updates the entry with data from the *Paging Update*.
removeMobile	Prepares the removal of this mobile entry.

Support Methods

pagingAbort	Called when paging was aborted.
handlePagingReport	Handler for kernel paging reports concerning this mobile. Starts and stops paging, buffers data for this mobile and starts the flushing thread.

Buffering During paging of a mobile it is not yet reachable and its exact position is not yet known. Thus the data for the mobile has to be buffered in the gateway proxy. Each mobile entry has its own buffer but unless the MEP all data packets arrive through one special raw socket, the paging socket. Moreover, buffering is done by the main thread. This is acceptable because the time period for which buffering is necessary is relatively short and so buffering will not be necessary for a lot mobile hosts at the same time. As in the MEP flushing is done by a separate thread to avoid interruption of message processing.

Buffer related Methods

startFlushThread	Starts the flushing thread for the buffered data packets.
flushThread	The flush thread is a static method that is passed a pointer to the corresponding mobile entry as an argument. Thus is can be logically treated as a non-static method. The flush thread inserts an *Ethertap* header to each packet and forwards it to the mobile host through the multicast channel. Flushing can be controlled by a flush policy.
flushThreadCleanup	Cleanup method for the flush thread.
insertEthertapHeader	Static method that does the insertion of *Ethertap* headers for the flush thread. The necessary room for the header is reserved on the buffering of the packet.

3.4.2 Methods

Message Handlers

handle_PagingSocket	Handler for kernel reports from the paging socket. Checks whether a corresponding mobile entry exists and either dispatches the report to this entry or reports failure to the kernel.
handle_PagingUpdate	Handler for *Paging Update* messages. Creates and updates the corresponding mobile entry in the mobile database.

Sending of Messages

send_PagingReq	Sends a paging request for the given mobile host to the given paging area.

Gateway-Specific Methods

insertMobileExpiry	Inserts an expiry timer for a mobile entry into the timer list.
insertPagingExpiry	Inserts an expiry timer for the paging process of a mobile into the timer list.
bufferPolicy	Calls the buffer policy handler if one exists. The handler is passed the IP header of the packet in question. It controls whether the packet should be buffered. If no policy handler is installed, all packets will be accepted.
flushPolicy	Calls the flush policy handler if one exists. The handler is passed the reception timestamp and the IP header of the packet in question and controls whether the packet should be forwarded to the mobile or discarded. All packet will be forwarded if no policy handler is installed.

Handling of Timers

mobileExpire	The lifetime of a mobile entry expired. The entry is removed unless we are paging or flushing. In this case the entry is prolongated until paging or flushing finishes.
pagingExpire	The paging process timed out.

3.4.3 Sending of Multicast Packets

PAGING_REQUESTs and buffered packets are multicast packets originating from the gateway, which is also a multicast router. Under Linux, sending of multicast packets from a multicast router is treated as if the router acted as a normal end system. On sending you have to specify a single interface on which a multicast packet should leave the host. In our case it is however desired that multicast packets are fed into the multicast routing mechanism the same way as forwarded packets are. This is achieved by a trick:

The *Ethertap* device, a software network device simulating an Ethernet adapter, is used. Everything that is written into a special socket from user space appears in the kernel as if it was received from the *Ethertap* network device. Everything that is sent to the *Ethertap* device appears on the socket. Only small modifications were necessary to allow an MTU of up to 65535 bytes (to avoid useless fragmentation) and to enforce acceptance of packets from *Ethertap*, which were originally expected from another interface.

This could better be achieved by divert sockets (see Sec. 3.5.1 for details). Unfortunately when I discovered the concept of divert sockets and their working implementation for Linux the agents had already been implemented.

3.4.4 Paging

Paging is done partly in the Gateway Proxy and partly in the kernel. The paging process from the view of the kernel is described in Sec. 4.3. This section discusses only the user space part:

1. The agent receives a `MFC_PAGING_CHECK` report through the paging socket.
   (a) If a corresponding mobile entry exists it will do the further processing. It sends a *Paging Request* and notifies the kernel that paging has started.
   (b) Otherwise this will be reported to the kernel and the packets will be treated like normal multicast packets.
2. From now on packets for the mobile host are send up to the agent through the paging socket and are buffered here. Packets to be buffered can be distinguished from kernel reports by having an non-zero protocol field (unless the kernel reports).
3. When a *Paging Update* with lifetime zero is received for the mobile host the kernel is notified that paging has finished. The kernel waits now for the corresponding multicast group to become existent.
4. This (or the failure) is reported by the kernel.
   (a) In the first case flushing is started.
   (b) In the second case the buffer is freed.

3.4.5 Management Interface

The management interface of the Gateway Proxy understands the following commands. For the syntax see the paragraph on `MB2_Parser`.

getState	Prints whether the agent was initialized (0/1).
getMobile	Prints the entry of the mobile with the given address or a table of all known mobiles finished by `end` in the following format: unicast IP address, deleted flag, paging area address, interface name, lifetime, registration id (high,low), expiration time (sec,usec), paging flag, paging_finish flag and flushing flag.
getPagingSeqno	Prints the last paging sequence number.
close	Close the connection.
terminate	Shutdown the agent.
reset	Shutdown and restart the agent.

3.5 Support Code

3.5.1 MB2_Socket Hierarchy

The access to network sockets is encapsulated by socket classes. The different types of sockets form a hierarchy of classes that inherit from the abstract `MB2_Socket` class (see Fig. 3.7). All sockets provide methods for opening, closing, reading, writing and querying about the amount of data available for reading. The most sockets can be passed certain flags on opening that depend on the type of the socket. Some sockets provide additional type-specific operations. Special features or implementation details of some socket types are discussed in the following paragraphs.

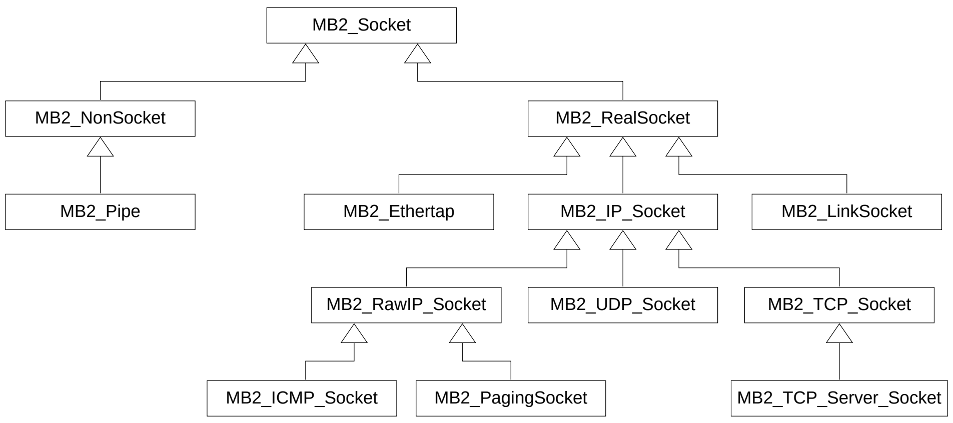


Figure 3.7: Class diagram: Socket hierarchy

Socket Filter Linux allows a packet filter to be attached to a network socket. The Linux socket filter model is essentially an in-kernel implementation of the Berkeley Packet Filter model[11]. The possibility to attach filters to sockets is somewhat limited in this implementation. It provides only the filtering by source, destination and protocol fields in the IP header and a special filter only accepting data packet, thus filtering out *MOMBASA SE* signaling, broadcast packets etc. The latter is used for idle detection in the mobile host. The filter programs were mainly generated by the tcpdump program and edited by hand afterwards.

Non-Sockets A subclass of `MB2_Socket` that does not represent a network socket is somewhat astonishing. Limiting inheritance is generally considered a design error. However, I decided to do just that in this case. The reason is the following: Some agents are multithreaded. It is necessary to signal certain conditions from sub-threads to the main thread (e.g. the Mobile Agent's idle thread has to signal a wakeup condition to the main thread). However, the main event loop only handles events from socket-like input channels and pre-scheduled timer events. The waiting for this kind events can be implemented quite easy and efficiently (without the need for busy waiting) on the basis of the UNIX select function (see the paragraph about `MB2_SocketGroup` for details). To avoid introduction of a third type of events the inter-thread communication is done via a pipe. To make the pipe insertable into a socket group it was implemented as a subclass of `MB2_Socket`.

The cleanest solution would probably have been to make pipes and sockets subclasses of a common base class (e.g. `MB2_CommChannel`) and make `MB2_SocketGroup` a group of such communication channels. Since all these classes are implemented on the basis of file descriptors this would also have been possible.

Real Sockets Classes that represent real network sockets are derived from the `MB2_RealSocket` class. Besides the inherited methods it provides the possibility to peek for data without removing it from the kernel buffers. It allows setting of a socket filter and retrieving the timestamp of received packets. Some general socket options such as binding to devices and broadcast can be set.

Ethertap The *Ethertap* device is a software network device emulating a Ethernet device. Frames can be sent to a *netlink* socket associated with the device and appear in the kernel as if they had been arriving from a network device. Frames sent by the kernel to the *Ethertap* network device appear at the *netlink* socket in user space. Thus user space programs can make packets appear as coming from outside. They can even emulate whole networks. The access to the *netlink* socket corresponding to the *Ethertap* device is encapsulated in a `MB2_Ethertap` which is a subclass of `MB2_Socket`.

Divert Sockets Another possibility to extract packets from the kernel are divert sockets. Packets can be diverted from the kernel protocol stack to the divert socket in user space and can be re-injected into the kernel via the divert socket. The concept comes from NetBSD but an experimental implementation for Linux also exists [1]. Unfortunately this became to my knowledge after the implementation of the *MOMBASA SE*. Divert sockets could have provided a uniform interface for buffering and flushing the buffers in MEPs and Gateway Proxy and for sending of *Paging Requests* in the Gateway Proxy.

Raw IP Sockets Raw IP sockets in Linux will only allow receiving if they are bound to a certain IP protocol. A raw IP socket for all IP protocol (as is necessary for buffering in the MEP) can only be used for sending. However the encapsulation of raw IP sockets as represented by the `MB2_RawIP_Socket` class does allow reception of all IP protocols. This is implemented by using a packet (i.e. link-layer) socket for receiving and a raw IP socket for sending. The packet socket is bound to IP and configured to strip the link-layer header. For the clients of `MB2_RawIP_Socket` this is totally transparent and it appears for them as if they are reading from and writing to the same socket.

Why not *libpcap*? The pcap library is a portable library used for capturing of network frames. This is used by the tcpdump program for example. However, the *libpcap* does its own buffering that is not intended to keep packets permanently. So I would still have had to do my own buffering which would have resulted in double-buffering. The *libpcap* reads the packets from the kernel into its own buffer and these packets would have had to be copied into the permanent buffers. This would have resulted in a unnecessary overhead.

3.5.2 MB2_SocketGroup

The socket group (which should better be a group of communication channels as mentioned in the paragraph about non-sockets) is not designed as a container for sockets but as a class to group them as a common source for events. Sockets can be members of multiple groups. `MB2_SocketGroup` is an object-oriented interface to the select function which allows a group of file descriptors to be watched. The group is used to wait for any socket in the group to become ready to read from or to write to. This waiting can be limited by a timeout. It is mainly used in the main event loops of the agents. Members of the group are the sockets on which signaling messages arrive and the timeout is set to the expiration of the next timer in the timer list.

3.5.3 MB2_Timers

`MB2_Timers` is a template class that is parameterized by the class of which the methods are called on expiration of the timer and the type of the argument of these expiration methods. The class represents a chronologically ordered list of timers. Each timer consists of the time at which it expires, the method to be called, the object for which the method is to be called and auxiliary data to be passed to the method. The timers are able to store times in microsecond precision, however the real precision depends on the granularity of system clocks and timers (Linux on a x86 architecture usually has a 10 milliseconds precision, i.e. 100 timer ticks per second). Times can be specified relative to the current time or absolute (which means relative to the “epoch” which begun on January 1, 1970). Methods for insertion and changing of timers return a handle that is implemented internally as a pointer to the corresponding list node.

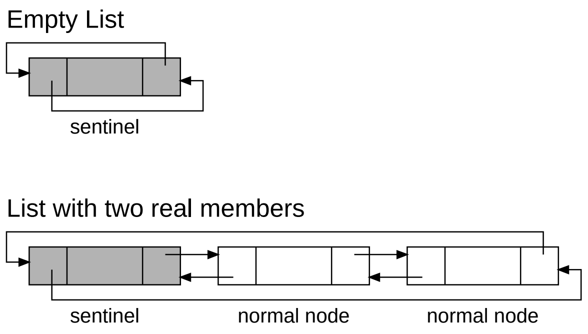


Figure 3.8: Linked list with sentinel

The list itself is implemented as a circular double-linked list that uses a sentinel as described in [2]. With a sentinel node, start and end of the list are not represented by NULL pointers but by a special node, called the sentinel. This avoids special handling of boundary conditions such as deletion of the first or last node. There are several reasons why a linked list and not a more complex data structure, such as a binominal heap, is used. Although most operations of heaps have a better asymptotic behavior than linked lists (see Table 3.1) the involved factor is a lot bigger and the implementation a lot more complicated. Thus the better asymptotic behavior would only become effective for timer lists with a lot of timers. Moreover an operation often performed is the deletion which can be performed in constant time with linked lists.

Operation	Linked List	Binomial Heap	Fibonacci Heap
Key Insertion	$O(N)$	$O(lg(N))$	$O(1)$
Key Deletion	$O(1)$	$O(lg(N))$	$O(lg(N))$
Key Changing	$O(N)$	$O(lg(N))$	$O(lg(N))$

Table 3.1: Asymptotic behavior of several data structures

3.5.4 MB2_Interface

The `MB2_Interface` encapsulates the access to network interfaces such as the retrieval of interface name, address, index, flags, etc. Data that usually remains constant is cached in the object instead of retrieving it from the kernel with IO-controls every time.

3.5.5 MB2_PolicyHandler

Most policy handlers are derived from the `MB2_PolicyHandler` template class. The template is parameterized by the argument and return types of the pre- and post-processing method and by the type of the property field. The agents provide type definitions of the handler with the correct argument types. To implement a policy handler one derives from these defined types and overrides the virtual pre and post methods.

The same module that provides the `MB2_PolicyHandler` template class also provides template functions and template classes for registering, de-registering and calling of policy handlers. The implementation of policy handlers employs the strategy design pattern (also called policy pattern).

3.5.6 MB2_Buffer

`MB2_Buffer` provides a ring buffer for variably-sized data objects such as IP packets. `MB2_Buffer` is not responsible for reading the data into buffer but only for reserving the memory space. Since the buffer is a ring buffer, it may happen that a data object is split into two parts with the first part being saved at the end of the buffer and the second part being saved at the beginning of the buffer (like packet 4 in Fig. 3.9). This depends on the ability to read from and write to gather/scatter arrays. Data read from sockets can be split into multiple buffers (i.e. scattered) and written to sockets from multiple buffers (i.e. gathered). The splitting point within the message is guaranteed to be aligned to a given alignment. The last reservation can be canceled (e.g. if a read error has occurred). New data can overwrite older data (remember that the buffer is cyclic), however, it is guaranteed that only whole data objects are discarded. For example if a packet bigger than the unused space in Fig. 3.9 is stored in the buffer packet 1 will be discarded. If the new packet is very big maybe even packet 2 will be removed from the buffer. To preserve the boundaries between data objects the size of the data objects is stored in front of the data.

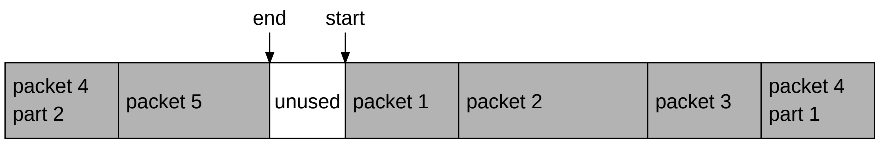


Figure 3.9: Ring buffer

The buffer can flush all data objects to a given socket. Each data object is written in a single operation (this will be important if the socket is datagram oriented). The flush method can be passed operations that manipulate the data object and to retrieve the address to which the data object before sending. These operations can also reject the data objects. The calling of flush policy handlers is usually done in such an operation.

3.5.7 MB2_BufferHandler and MB2_FlushHandler

`MB2_BufferHandler` and `MB2_FlushHandler` are the only policy handlers that are not derived from the `MB2_PolicyHandler` template. They only provide a virtual function operator that should be overridden in real policy handlers. The buffer policy handler is passed the IP header of a packet to buffer, the flush policy handler additionally gets the receiving timestamp of the packet. They will return `true` if the packet shall be buffered/flushed respectively and `false` otherwise.

3.5.8 Multicast Channels

The *MOMBASA SE* was designed to work with any kind of IP multicast. It was designed especially with Single Source Multicast in mind. However, since no free SSM implementation for Linux exists, this is not implemented. The `MB2_MC_Channel` class provides an interface to Single Source Multicast channels but discards silently the source address. It uses socket options provided by the Linux kernel for joining and leaving multicast groups. The kernel maps these socket options to IGMPv2 messages and takes care of the multicast leaf signaling. To support SSM, which uses an extended socket interface, the methods of `MB2_MC_Channel` would have to be rewritten but the rest of the *MOMBASA SE* could stay the same.

3.5.9 MB2_Route

`MB2_Route` represents a network route. It provides methods for setting, modifying and querying of the correspondent routing table entry in the kernel. Access to the routing table in Linux is not very well documented. The `MB2_Route` class is more or less an object-oriented variant of the corresponding support code of Dynamics Mobile IP [13] which is based on the *libnetlink* by Alexey Kuznetsov.

3.5.10 MB2_Parser

`MB2_Parser` implements a parser for simple configuration files or similar things (e.g. the management interface). The `MB2_Parser` hierarchy (see Fig. 3.10) employs the template method design pattern. The reading of text lines is not implemented in the base class but delegated to the subclasses. `MB2_FileParser` implements reading from a text file and is used for the parsing of configuration files in MOMBASA SE. `MB2_TCP_Parser` reads from a TCP socket. It is used for the management interface.

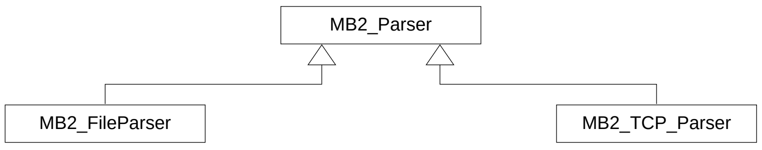


Figure 3.10: Class diagram: Parser hierarchy

The parsed (very simple) language has the following properties:

- Empty lines and comments introduced by # are ignored.

- Other lines contain a keyword and may contain up to 16 arguments. Starting and trailing whitespaces and trailing comments are ignored.
- Keywords are case insensitive.
- The keyword is separated from the arguments by colon or an equal sign and/or one or multiple whitespaces.
- Arguments are separated by commas and/or whitespaces.

The parser is passed a table that associates a numeric token and the minimum and maximum number of arguments to each keyword. The client of the parser can advance line per line, read the keyword token, the number of arguments and the arguments themselves in various formats (strings, integers, boolean value, IP address, pair of IP addresses).

3.5.11 MB2_Signal

`MB2_Signal` implements a template class to enable object methods to be installed as UNIX signal handlers. The template can be parameterized by the class of which the methods should be called and the type of an auxiliary argument that can be passed to this method.

It must be guaranteed that only one instance of a `MB2_Signal` class exist in an application. Thus the template is implemented as a singleton. However, the scheme described in Sec. 3.1 is not sufficient for a template class, since a static member would be created for each incarnation of the template. To make it work nevertheless, I instantiated the template with dummy parameters (`MB2_Dummy` as the receiver class and `bool` as the data type). Only the static instance pointer of this class is used. The constructor of a `MB2_Signal` class checks that this pointer is `NULL` and assigns its own address to the pointer. To make it assignable, the pointer to the instance is untyped (i.e. `void`).

The rest of the implementation is straightforward. Since the number of signals is constant and quite small a static array of signal entries is sufficient. A static method of the template class is installed as the signal handler for each signal for which a special handler is installed. This handler calls the specified method of the specified object with the signal number and auxiliary data as arguments.

3.6 Documentation

The declarations in the source code are commented in a format suitable for doxygen 1.2.10 (it works also with 1.2.3). Doxygen is a tool that generates HTML and other documentation from the source code. A configuration file is enclosed in the *MOMBASA SE* distribution. To generate the documentation
`doxygen doxygen.cfg` must be called in the source directory of *MOMBASA SE*.

Chapter 4

Kernel Implementation

4.1 Navigating in the Kernel

Network Address Translation from and to multicast addresses and paging requires modifications to the kernel. Navigating in the Linux kernel can be quite confusing. A useful tool for browsing the Linux kernel is LXR [10]. Nevertheless, it is not an easy task to understand the operation of the kernel. To avoid getting lost one should select a certain scenario one is interested in and follow the control flow of only this scenario, ignoring exceptional cases at first. Even this is not as easy as it sounds, as the Linux kernel is more or less object-oriented. However, it is not implemented in C++ but object-orientation is emulated in C by putting pointers to functions into data structures. Often these functions are passed a pointer to just this structure as an argument. Note that this is equivalent to virtual methods in C++ where it is hidden from the programmer that the methods are called through pointers and the object is passed as an implicit argument (called `this` in C++).

In the case of function calls through pointers it is often easier to do an educated guess which function is called and assert later that the variable really pointed to this function. For example suppose we are looking at the function `sock_recvmsg` (taken from Linux 2.2.18):

```
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size,
                 int flags)
{
    struct scm_cookie scm;

    memset(&scm, 0, sizeof(scm));

    size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
    if (size >= 0)
        scm_recv(sock, msg, &scm, flags);

    return size;
}
```

This call to `sock->ops->recvmsg` is really ugly. However, if you know that the socket for which it is called is a UDP socket, it is most probable that the function called is `udp_recvmsg`. Such heuristics can be used in many cases. It also may be helpful to look at initialization code of certain kernel parts to figure out how those pointers to functions are usually initialized.

4.2 Multicast Network Address Translation

To make the usage of multicast transparent to fixed and mobile host, packets must be translated from unicast to multicast in the gateway and back in the access point. However, the Linux kernel only supports network address translation between unicast addresses. Thus the standard kernel (2.2.18 in our case) had to be modified.

Besides the configuration files, three files had to be modified: route.c, ip_forward.c, ipmr.c and fib_frontend.c. All of them are in the net/ipv4 directory.

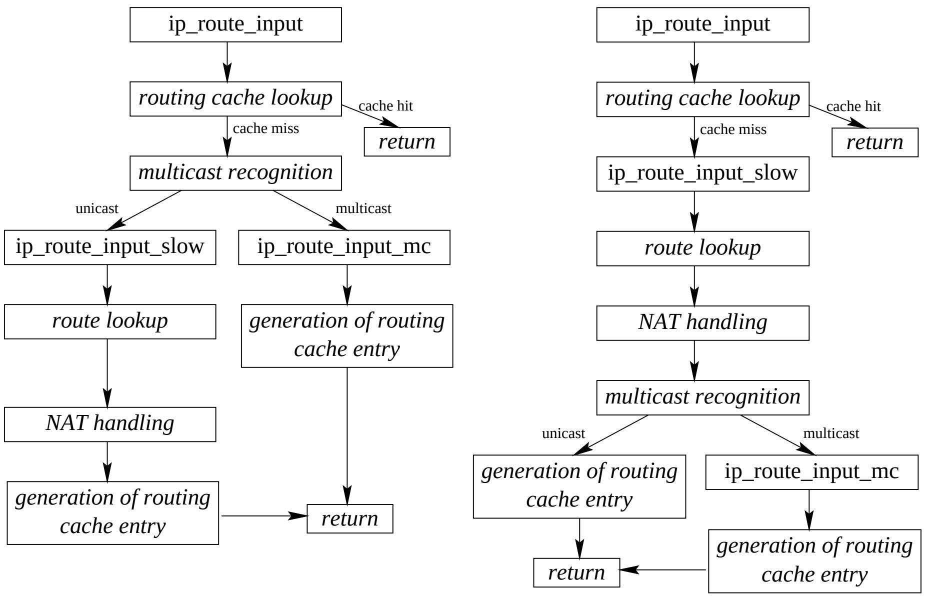
In this paragraph I describe the control flow in the unmodified kernel (see Fig.4.1(a)). During procession of an incoming packet *ip_route_input*(route.c) is called for retrieval (or generation) of a fitting route cache entry. First, this function tries to find an entry in the routing cache, which matches the packet's source, destination, input interface and type of service. If it finds one it will return successfully, otherwise a cache entry must be generated. In the next step unicast and multicast traffic is separated. For unicast traffic *ip_route_input_slow*(route.c) is called, for multicast traffic *ip_route_input_mc*(route.c). The former does a route lookup and creates an appropriate routing cache entry taking translation of source and destination address into account. The latter does not use the unicast routing tables and thus ignores NAT entries. The address translation itself happens in *ip_forward*(ip_forward.c). If the NAT flag is set in the packet's routing cache entry *ip_do_nat* (ip_nat_dumb.c) is called which changes the IP addresses in the IP header and recalculates checksums for IP, TCP, UDP and ICMP.

When addresses can be translated between the multicast and the unicast realm, the major problem is that we can't tell from the original address if a packet has to be treated as unicast or multicast. So the multicast recognition has to be deferred until we know the mapped address (or that the packet doesn't have to be mapped).

The following description assumes that both translation from multicast and to multicast addresses is enabled: After routing cache lookup *ip_route_input_slow* is now called no matter if the destination address is multicast or not. A lookup of the (unicast) routing table is done. If the destination address is multicast and no fitting NAT entry has been found *ip_route_input_mc* is called as before. If a NAT entry exists the mapped address is checked whether it is multicast or unicast and depending on that unicast or multicast routing is done.

Some minor change were applied to *ip_route_input_mc* to set the mapped addresses and the NAT flag in the routing cache entry. In *ip_mr_input*(ipmr.c) a call of *ip_do_nat* was added since it existed only in the unicast branch before. Finally, the check in *ip_forward* that the packet type was `PACKET_HOST` had to be relaxed since the original packed could also have been of type `PACKET_MULTICAST` before mapping.

In the unmodified kernel, for an incoming packet it is checked that a unicast route to the source of the packet exists. This causes upstream packets from the mobile to be dropped in the gateway since there is only a NAT entry from unicast to multicast. Therefore *fib_validate_source*(fib_frontend.c) was modified to check whether a NAT entry to multicast exists for the checked packed. However if reverse path checking is enabled (i.e. the check that a packet came from the interface replies woulde be sent to) the packet will be dropped nevertheless.



(a) Unmodified kernel (b) Modified kernel

Figure 4.1: Input routing in the Linux kernel

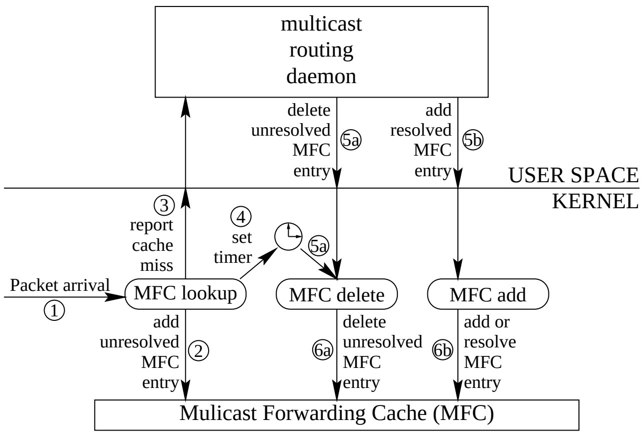
Network Address Translation is only appropriate for classical IP multicast. For single-source-multicast the source address would have to be changed to the address of the gateway. Since original source address must be restored in the access point this could only be done by IP encapsulation. Some modifications to the tunneling code would be necessary to be able to tunnel packets through multicast trees.

4.3 Paging Support

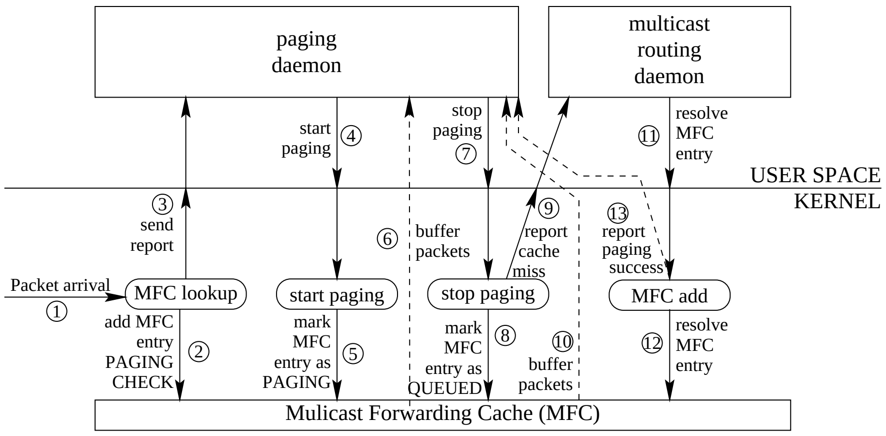
When data is sent to an inactive mobile host, it must be paged by the gateway. The paging itself is done by the gateway proxy or paging daemon (see Chapter 3.4.4). However paging is triggered by the reception of packets for the mobile which requires some kernel support.

Since the unicast packets for the mobile host are translated to multicast packets in the gateway the best place to insert paging support is the multicast forwarding cache (MFC) implemented in net/ipv4/ipmr.c. In the following paragraphs routing of multicast packets in an unmodified router will be described. After this the original paging support and why it wasn't appropriate is discussed. Finally the paging support as it is now is depicted.

In an unmodified multicast router (see Fig. 4.2(a) and Fig. 4.4(a)) when a multicast packet first arrives (1) an unresolved entry for the combination of source and destination is inserted into the MFC (2) and the cache miss is reported to the multicast routing daemon via the *mroute_socket* (3). A timer is set for the MFC entry (4). For a non-existent multicast group the timer will expire or the MFC entry will be deleted explicitely by the multicast daemon through the socket option `MFC_DEL` (5a, 6a). Otherwise the MFC entry will be resolved with the socket option `MFC_ADD` (5b, 6b). As long as the entry is not resolved (i.e. state `MFC_QUEUED`) arriving packets are queued in the unresolved entry as long as there are no more than 4 packets in the queue. The resolution of the entry among other things sets the output interfaces for this source/multicast group combination. The buffered and subsequent packets are only treated by the kernel and sent to all outgoing interfaces stored in the MFC entry.



(a) without paging



(b) with paging

Figure 4.2: Interaction between kernel, multicast routing and paging daemon

When the mobile host is inactive, the corresponding multicast group should not exist. Originally, the reception of a multicast packet for a non-existent group in the range of pageable groups was signaled by the kernel to the gateway proxy via the *paging_socket*. In the case that a paging cache entry existed for the mobile, the gateway proxy sent a paging request to the last reported paging area otherwise it deleted the MFC entry. During the paging phase the kernel sent all packets for the mobile up to the gateway proxy for buffering. When the multicast group was created, this was again signaled to the gateway proxy which flushed the buffer for the mobile and sent out all packets.

However under some circumstances a multicast group for an inactive mobile host could exist. This happens in the following case (see Fig. 4.3): When a mobile host sends its first inactive registration to another access point than its last active registration and doesn't send a deregistration to the old access point the old access point doesn't know about the inactive state of the mobile host and will stay in the multicast group until the lifetime of the mobile registration will expire. With the just described scheme, during this period paging would not be triggered and the mobile host would be unreachable.

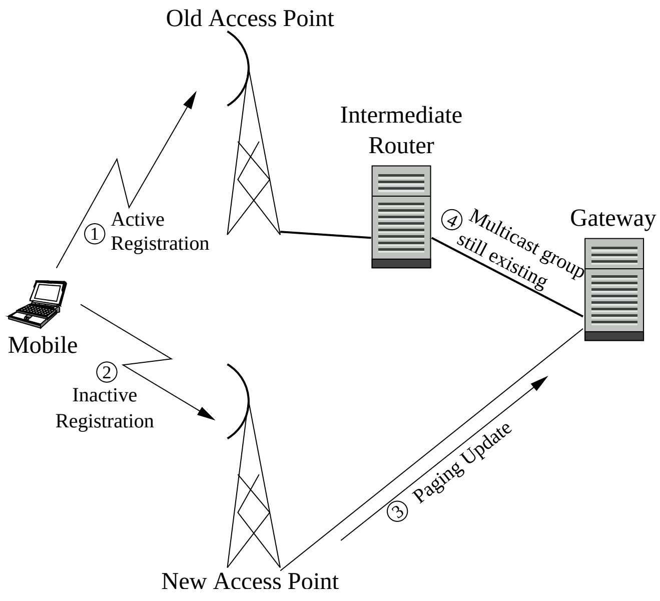
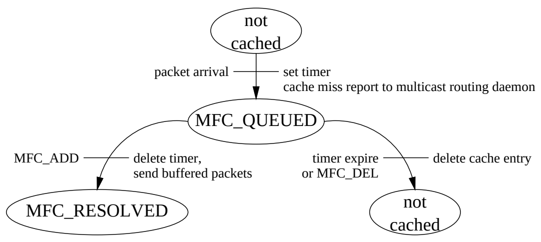


Figure 4.3: Existing multicast group dilemma

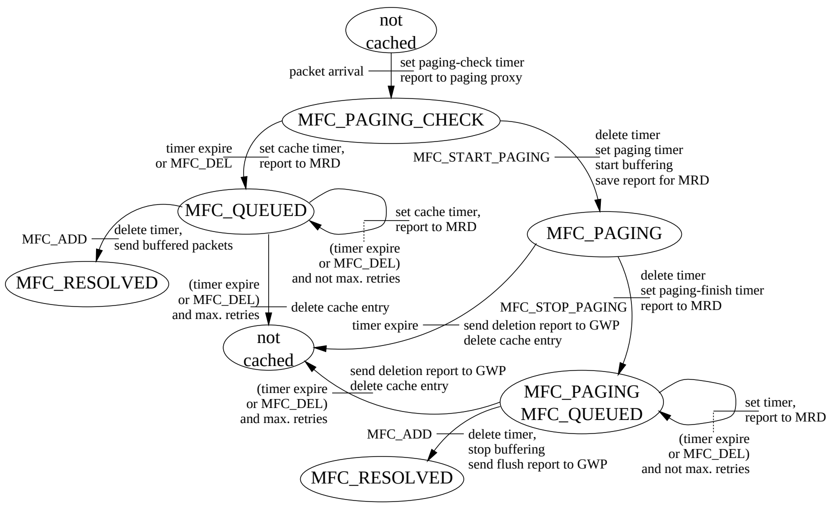
With the new scheme we don't depend on the existence or non-existence of the mobile-related multicast group but on the existence of a paging cache entry in the gateway proxy. Thus we have to notify the proxy first. The paging for an inactive mobile is depicted in the following paragraph (see Fig. 4.2(b)). Packets for multicast groups that don't correspond to mobiles are treated like before. Normal multicast routing is not affected by the paging patch. Entries of mobile-related (i.e. pageable) groups are marked with the `MFC_PAGEABLE` flag. This flag is ommitted in Fig. 4.4(b) although present in every shown state.

After the arrival of a multicast packet to a pageable group (1) a MFC entry with the flag `MFC_PAGING_CHECK` is created (2) and a timer is set. A report is sent to the gateway proxy via the *paging_socket* (3). If an entry exists in the daemon's paging table it will start paging and signal this condition to the kernel with the `MFC_START_PAGING` socket option (4). The MFC entry is marked as paging (5), data packets to the multicast group are now sent up to the gateway proxy for buffering (6). When the paging cache entry in the daemon was deleted through a paging update with lifetime 0, denoting that paging was successful, this is again signaled to the kernel with the `MFC_STOP_PAGING` socket option (7). The MFC entry is marked as unresolved (i.e. `MFC_QUEUED`) (8) and a previously saved cache miss report is sent to the multicast routing daemon (9). Data packets are still buffered in the gateway proxy (10). The resolution of the MFC entry by the multicast routing (11/12) is signaled to the paging daemon (13). Buffering is stopped and the buffers are flushed. The MFC entry is set to `MFC_RESOLVED`.

The real behavior is, however, even a bit more complicated since the previous description omitted some exceptional cases (e.g. timeouts and retries). For all details please consult Fig. 4.4(b).



(a) without paging



(b) with paging

Figure 4.4: State machines for Multicast Forwarding Cache entries

Chapter 5

Extensibility of the MOMBASA SE

5.1 Support of Generic Multicast

The actual version of the *MOMBASA SE* works with IP multicast as it is defined by the standard IP multicast service model in RFC 1112 [3]. The MEPs use IGMPv2 for multicast management. Since IGMPv2 works with all multicast routing protocols according RFC 1112, the *MOMBASA SE* is independent of the multicast routing protocol. However, there are multicast routing protocols which are less suitable for mobility support (such as broadcast- and prune protocols, like DVMRP [14] and some which are more suitable (such as PIM-SM [4, 5] which is based on a concept of rendezvous points).

Recently, multicast has been subject of research efforts. There are several proposals for multicast which break with the standard IP multicast service model. An example is single source multicast, such as EXPRESS [9] and PIM-SSM [6]. We understand the *MOMBASA SE* as a generic platform to investigate multicast-based mobility support. Hence, it is prepared for *Single-source multicast*, its usage requires only minor modifications of the implementation, i.e. of the `MB2_MC_Channel` class.

Moreover, *MOMBASA SE* has been designed to be suitable for a *generic multicast* which is based on a group identifier[1] and a member identifier [2] For generic multicast, a number of basic operations can be identified: Creation of group, subscribing to the group, un-subscribing from the group, destruction of group. The implementation of *MOMBASA SE* has been designed with respect to support generic multicast, and hence it is expected to be extended easily.

5.2 Support of Policies

Policies are rules to control and fine-tune certain behavior of the MOMBASA SE system. The usage of policies in the MOMBASA SE ensures a flexible and easy extensibility by certain functionality. Typical examples of policies are to control in the mobile when to do handover and which access point to select, to determine in MEP and Gateway Proxy which packets to buffer and which packets to flush from the buffer, to pre- and post-process protocol messages. Hence, the MOMBASA SE provides hooks for policy handlers

[1]With standard IP multicast this is a *class D* IP address.

[2]With standard IP multicast the unicast IP address of the member.

- that may select time and destination of handovers thus allowing evaluation of different handover schemes including predictive handovers.
- that control buffering of packets and flushing of buffers allowing evaluation of various buffer strategies.
- that may retrieve the signal quality of messages received from the last hop for use by other policies, e.g. the handover decision.
- that may pre- or post-process any protocol message of the environment to achieve goals that have not thought of.

5.3 Portability

In the *MOMBASA SE* only standard libraries are used. This facilitates porting MOMBASA SE to other architectures. In particular, it is feasible to port MOMBASA SE to handheld PC architectures running with a Linux operating system. Since memory is a scarce resource of handheld computers, it is important to have only a few dependencies.

5.4 Open Issues

There are still a few missing features:

1. The multicast routing daemon used in the testbed had the following limitation: Multicast senders must be in the same IP network as the upstream interface of the gateway, since the daemon does not support operation as a border router.
2. There are only a few policies implemented. The following policies are included in MOMBASA SE: A `SelectBS` policy that does a regular ping-pong handover and also supports externally triggered handovers (e.g. by the management interface) and a flush policy that only flushes packets that are younger than a configured number of microseconds. Additional policies can be easily implemented.
3. The current version of MOMBASA SE uses the standard ARP mechanism to resolve IP addresses to hardware addresses. This could be avoided by supplying the hardware address in MOMBASA signaling messages.
4. In the current version of MOMBASA SE, the support for Single Source Multicast is only implemented as stubs. The implementation of `MB2_MC_Channel` would have to be modified and a patch to the Linux kernel to support IP tunneling trough multicast tunnels would have to be developed to achieve full support.
5. Support of generic multicast as discussed in the previous section.

Bibliography

[1] Ilia Baldine. Divert Sockets for Linux. `http://www.anr.mcnc.org/~divert/index.shtml`, 2001.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 11, pages 204–208. The MIT Press, eighteenth printing edition, 1997.

[3] S. Deering. Host Extensions for IP Multicasting. RFC 1112, August 1989. `http://www.ietf.org/rfc/rfc1112.txt`.

[4] S. Deering, D. Estrin, D. Farinacci, M. Handley, A. Helmy, V. Jacobson, L. Wei, P. Sharma, and D. Thaler. Protocol Independent Multicast-Sparse Mode (PIM-SM): Motivation and Architecture. Internet Draft work in progress, October 1994. `http://citeseer.nj.nec.com/373495.html`.

[5] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. RFC 2362, June 1998. `http://www.ietf.org/rfc/rfc2362.txt`.

[6] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). Internet Draft work in progress, March 2001. `http://www.ietf.org/internet-drafts/draft-ietf-pim-sm-v2-new-03.txt`.

[7] A. Festag and L. Westerhoff. Protocol Specification of the MOMBASA Software Environment. Technical Report TKN-01-014, TKN, TU Berlin, Berlin, Germany, May 2001. `http://www-tkn.ee.tu-berlin.de`.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996. German translation.

[9] H.W. Holbrook and D.R. Cheriton. IP Multicast Channels: EXPRESS Support for Large-scale Single-source Applications. In *Proceedings of ACM SIGCOMM 1999*, pages 65–78, MA, USA, 1999. `gregorio.stanford.edu/holbrook/express/`.

[10] LXR - Cross-Referencing Linux. `http://lxr.linux.no`.

[11] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference (San Diego, CA, Jan. 1993), USENIX*. Lawrence Berkeley Laboratory, January 1993.

[12] University of Southern California. USC pimd. `http://catarina.usc.edu/pim/`, 2000. PIM-SM Version 2 Multicast routing daemon.

[13] Helsinki University of Technology Dynamics Group. Dynamics - HUT Mobile IP. `http://www.cs.hut.fi/Research/Dynamics/`.

[14] D. Waitzmann, C. Patridge, and S. Deering. Distance Vector Multicast Routing Protocol (DVMRP). RFC 1075, November 1988. `http://www.ietf.org/rfc/rfc1075.txt`.

[15] L. Westerhoff and A. Festag. Implementation of the MOMBASA Software Environment. Download at `http://www-tkn.ee.tu-berlin.de/research/mombasa/download/mombasa_se_1.0.tgz`.