A PROTOTYPE FOR POLICY DRIVEN CONTROL OF HETEROGENEOUS NETWORK ACCESS

Sébastien Pierrel, Topi Erlin and Janne Roslöf
Turku Polytechnic, School of Telecommunications and e-Business
Sepänkatu 1, FIN-20700 Turku, Finland
sebastien.pierrel@ericsson.com, topi.erlin@exrei.fi, janne.roslof@turkuamk.fi

Tony Jokikyyny and Hannu Lehtinen
Oy LM Ericsson Ab
FIN-02420 Jorvas, Finland
tony.jokikyyny@ericsson.com, hannu.lehtinen@ericsson.com

ABSTRACT

Internet has been continuously growing and the new dimension brought by wireless mobility will support this growth even in the future. Although the present technology supports operators to offer different kinds of wireless networks to end-users, most mobile terminals are yet capable neither to access simultaneously several networks nor truly roam between the different ones. Continuous research on this field has been ongoing for several years and it is just a matter of time when the majority of the mobile nodes will integrate more than one wireless access technology. Thus, a natural question is how these interfaces can be controlled and selected in such a way that they enable the best possible access – depending on the available resources, the operators' preferences and the needs of the end-user. In this paper, a prototype implementing a policy selection mechanism for the heterogeneous network access selection is presented.

KEYWORDS

Network access, Wireless communication, Mobile IP, Linux, J2ME

1. INTRODUCTION

Future mobile nodes usually integrate more than one wireless technology, so they could be connected to data networks by all possible ways. For example, a wireless terminal could open a packet switched access through WLAN, GSM-GPRS, UMTS-GPRS or Bluetooth. This scenario is illustrated in *figure 1*.

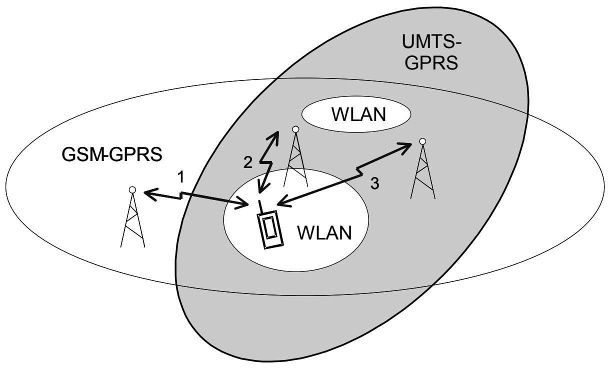


Figure 1. A mobile node located in area where there are three available access networks.

There is no network interface that could assure global scale of connectivity, and the different interfaces have very different characteristics. The available access interfaces may vary, for instance, in terms of cost, bandwidth and availability. It would be ideal to combine the best sides of each interface, like the GSM's high coverage and availability, as well as the WLAN's high bandwidth. When the currently used interface is, for example, losing its bandwidth, the device should be able to switch to another interface in the area without any action by or, eventually, even any notable effect to the end-user (seamless vertical handover). Moreover, also the operators may want to move some of the clients to another interface during high traffic load in a specific interface. In a situation when there are several interfaces available, the end-user might prefer to affect according to which policy (Yavatkar *et al.*, 2000) the used interface is to be selected.

2. BACKGROUND

Mobile IP is an open standard, defined by the Internet Engineering Task Force (IETF) Request for Comments (RFC) nr. 2002 (Perkins, 1996), latterly revised as RFC 3220 (Perkins, 2002). The mobile IP is a mechanism for maintaining transparent network connectivity to mobile hosts. A mobile host can be addressed by the IP address it uses in its home network (home IP address) regardless of the network that it is currently physically attached to. Therefore, the ongoing network connections to a mobile host can be maintained even as the mobile host is moving from one subnet to another. The main concepts of the Mobile IP are illustrated in *figure 2*.

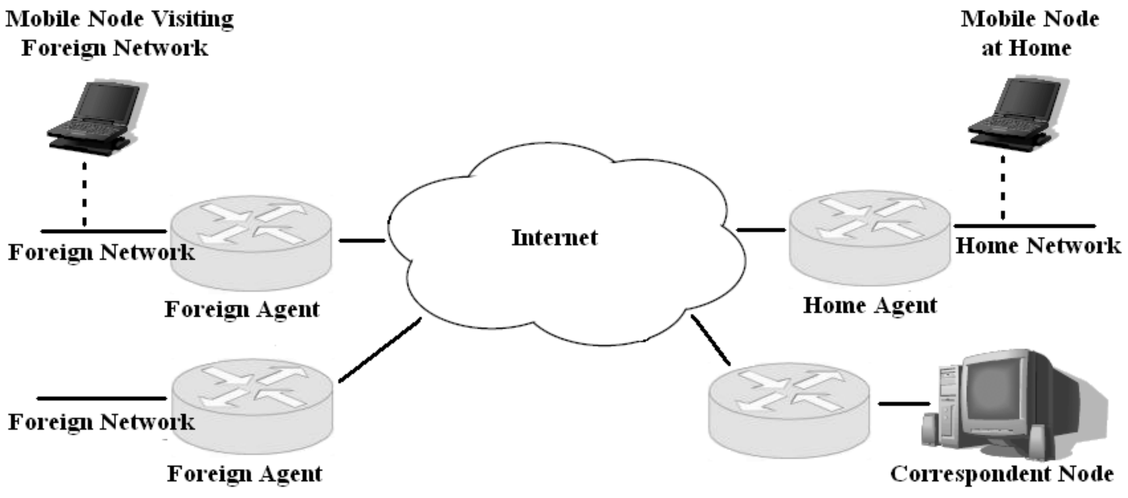


Figure 2. The main components of the mobile IP solution.

Home address is a permanent IP address that is assigned to a mobile node (MN). It remains unchanged regardless to where the mobile node is attached to the Internet. Whenever the MN is not attached in to its home network, the home agent gets all the packets destined to the MN and arranges to deliver them to the MN's current point of attachment using the

MN’s current care-of address. The foreign agent is a router on an MN’s visited network providing routing services to the MN while registered. The foreign agent extracts the datagrams that were tunneled by the MN's home agent and delivers them to the MN. (Yi-an, 1995; Perkins, 1996)

Mobility support in IPv6 follows the design of mobile IPv4. It retains the ideas of a home network, home agent, and the use of encapsulation to deliver packets from the home network to the MN's current point of attachment. While discovery of a care-of address is still required, an MN can configure its care-of address by using stateless address auto-configuration and neighbor discovery. Thus, the foreign agents won’t have to support mobility in IPv6. (Perkins, 1996; Deering and Hinden, 1998; Arkko, Johnson and Perkins, 2003)

MIPL Mobile IPv6 for Linux is an implementation of the mobility support in IPv6 developed by the Helsinki University of Technology (HUT) (Tuominen and Petander, 2001). The MIPL stack has been released under GNU Public License (GPL) and is available for anyone for free (www.mipl.mediapoli.com). A joint project between Ericsson Corporate Research and HUT implemented a multi-access extension to the MIPL and a GUI using which a user can control the interface selection. The Linux IPv6 stack and the MIPL were both originally designed to use only one network interface at a time. Thus, the IPv6 stack was modified so that it could operate with several simultaneous network interfaces and routers. Also the MIPL was enhanced to support multiple interfaces. (Jokikyyny *et al.*, 2002; Jokela *et al.*, 2002)

The simultaneous multi-access feature was implemented by updating the IPv6 routing table and the MIPL binding update list based on the user defined policies and the status of the network interfaces. For example, the user was able define a policy that has the destination address of his e-mail server, and list of interfaces. This causes the following: A routing table entry for the e-mail server is created and the first priority interface on the list of interfaces is set as the outgoing interface for this route. Also the next-hop address is set to be the IPv6 link-local address of the router found on the first priority interface. The MIPL binding update list is updated in order to have the entry for the destination address, and the used interface. After that a binding update is sent to the e-mail server, where the care-of address of the first priority order interface is used and the message is sent through that interface. This causes the return packets to come through the same interface. (Jokela *et al.*, 2002)

However, the implementation did not enable transparent vertical handovers, i.e. the user was forced to do them manually. The goal of the project presented in this paper was to develop the prototype further to make these handovers to happen automatically, so that the vertical handovers would be seamless, and to create a policy based tool for selecting the best interface to be used in heterogeneous wireless environment.

3. PROTOTYPE ARCHITECTURE

The prototype was divided in two main parts: An API module that implements the policy selection mechanism, and a GUI application that gives the user the possibility to define policies and to monitor the selection of the network access. The different architectural entities in the prototype environment are illustrated in *figure 3*. A similar concept could be used in a real mobile terminal.

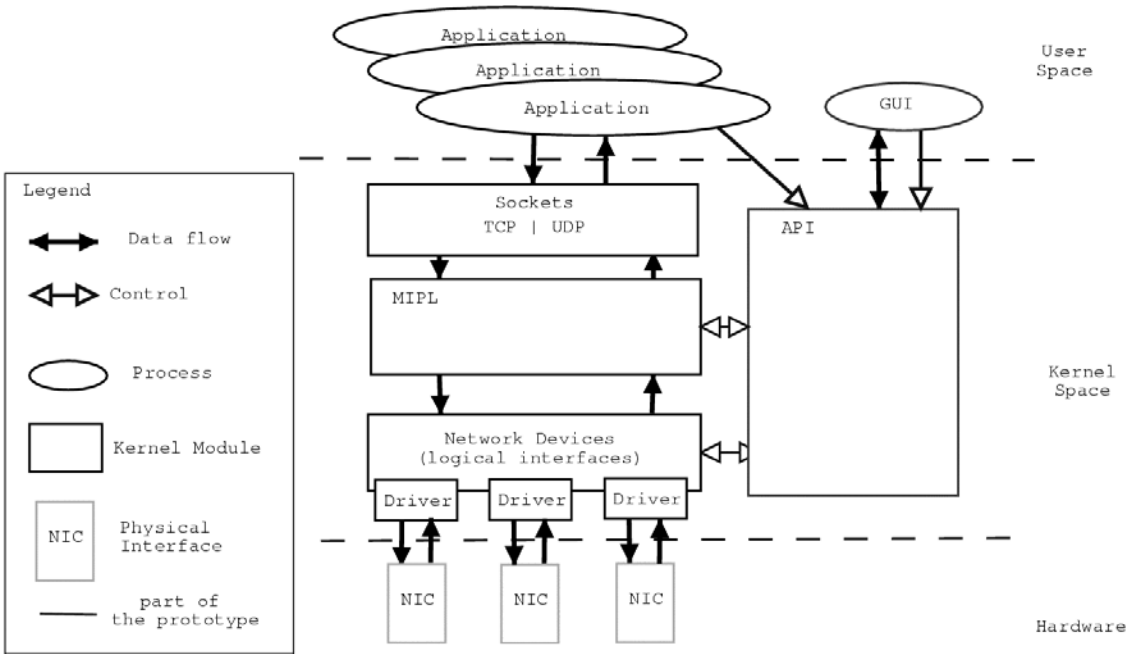


Figure 3. The prototype architecture.

The purpose of the API module is to add an interface selection mechanism to the MIPL stack. It sorts the priority ordered interface list, according to default policy sent by the GUI, and forwards this list to the stack. It can fetch data from the stack and deliver it to applications, and vice versa. Combining the policies given by the applications and the availability of the interfaces from network devices, the API module defines the preferences of interfaces. The GUI basically gives the policy rules to the API but other applications have the possibility to define their own policies, too. The API module was implemented with the C programming language, and designed to run in the kernel of the Linux operating system.

The GUI is used for sending user selected policies to the API module, so that the module can take into account the user preferences while enabling and disabling interfaces in Linux kernel space. It also displays interface details (e.g. operator, cost and bandwidth information) to the user. The GUI was implemented with Java 2 Micro edition's (J2ME) MIDP 1.0. Thus, it is mainly targeted to mobile phones with proper Java support.

The policy rules define how to sort the list of interfaces into a priority order. The policy data is the source containing data about different access interfaces provided by specific operators. The policy data could, for instance, be located in the Internet and the content in it could change continuously. Whenever the information changes, it should be updated automatically to the MN using, for example, WAP's PUSH method. If the policy rules would be in the Internet, the operators could easily affect which access interface clients must use – or allow the clients themselves to take the decision. The rules and data could also be stored in the terminal only, but in that case the operators would not have any influence to client's type of access unless the data would be hard coded, for instance, in the user's SIM card.

The hard coded options do not provide flexibility. The automatic policy data updating would be nice but it would not be so simple just to update data automatically without user's permission; usage of bandwidth costs money. In this prototype the data is stored in an XML file and updated manually via the GUI.

4. THE STRUCTURE OF THE API MODULE

The API can be divided in three main functions: Communication with user space, network interfaces management, and interface selection mechanism. This structure is illustrated in *figure 4*.

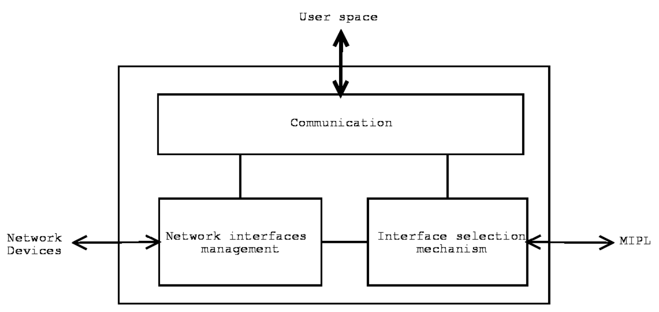


Figure 4. The organization of the API module.

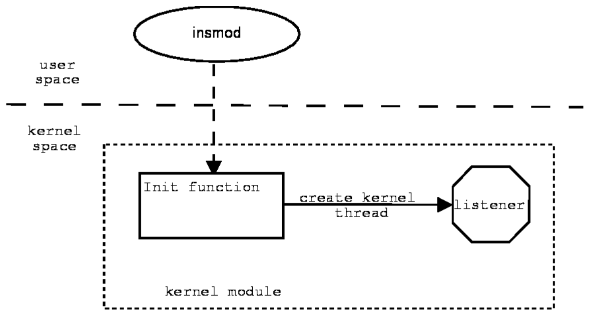


Figure 5. Creation of a kernel thread for listening to a socket.

The API module needs to be able to send information to applications at any time. Therefore, applications need to register to a connection-oriented service. As this prototype focuses on being interfaced by a GUI made with J2ME, many limitations in the communication between the kernel module and the GUI exist. Therefore, the communication was implemented using TCP sockets. At initialization, the module starts a kernel thread that will listen to a TCP port and wait for connections of clients (see *figure 5*).

Since the API has to give asynchronous notifications to applications, keeping track of all connections made concurrently is obviously needed. Two kinds of connections are to be taken care of: pipes and sockets. To handle the communication properly, the API module registers a character device file. A linked list `connector` is used to register every new connection. A system call is executed in the context of the process performing it, so no specific process is needed in the kernel. Whenever a process opens the device file, its process identifier (PID) is registered to a `connector`. When the listener creates another kernel thread to handle the connection, the `kthread` structure pointer is registered. The connector structure is illustrated in *figure 6*.

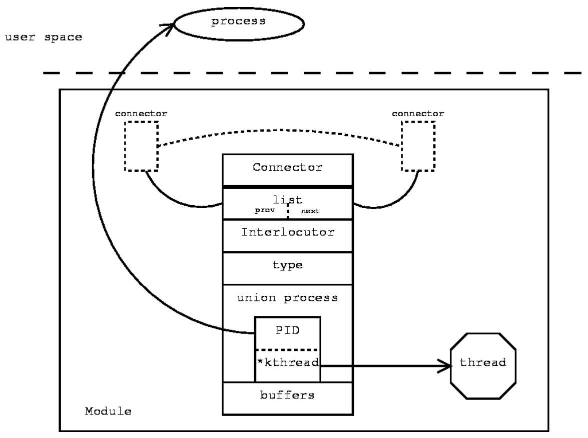


Figure 6. The `connector` structure.

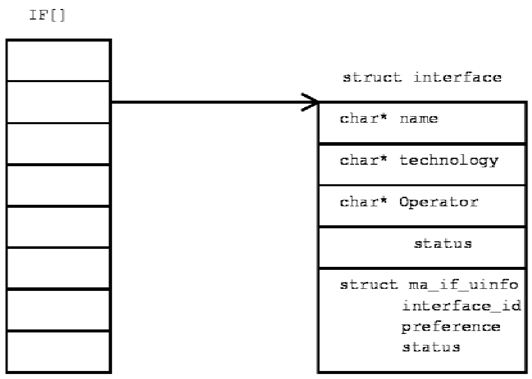


Figure 7. The `interface` structure.

The network interface management is articulated around the `net_device` structure, from which the status of the interface can be fetched. More information, such as the available operators, bandwidth, delay and other QoS features, is requested. A protocol on layer 2 is required to provide such information. Since such a protocol does not exist, those features were simulated in this prototype. The module contains an array of structures containing data related to the interfaces (see *figure 7*). The fields `name`, `technology`, `Operator` and `status` match with the policies given by the user space application. The structure `ma_if_uinfo` is used by the multi-access feature of MIPL to set the priorities. This structure has to be passed into the stack when applying changes to the priorities.

The API saves the policies in the memory as shown in *figure 8*. The policies are stored in a double-linked list filled with information given by the applications and the GUI. The default policy is entered at the tail of the list. The field `Policy_name` is the identifier of the policy. The field `criteria` is a structure that triggers the use of a policy over the default one. The criteria are given as parameters by the applications when defining the policy. The default policy has no criteria. The array `IF_priority` lists the interfaces available for the policy in order of preference. Since the applications, and in particular the GUI, are not aware of the interface names in the system (e.g. `eth0`, `eth1`, `ppp0`), the policies base the identification of an interface on the specified access technology and the operator name attached to each interface. The field `rule_name` is relevant to the application that has defined the policy. Thus, a profile can be retrieved later on by the owner of the policy without transmitting the priority list.

When an event (such as a new connection, a change in the status of an interface, or a new policy) occurs, the list of policies must be refreshed to redefine the priorities and, accordingly, the selection of the interface for the connection. As a kernel module, the MIPL stack exports functions to the kernel. That is, the API module can directly call those functions. The function `ma_ctl_set_preference` lets a user program change the priority of an interface via an IOCTL call. Although the API module allows applications to define policies in addition to the default policy, in this prototype only the default policy is applied to the system. The module modifies the routing table of the MIPL according to the active default policy.

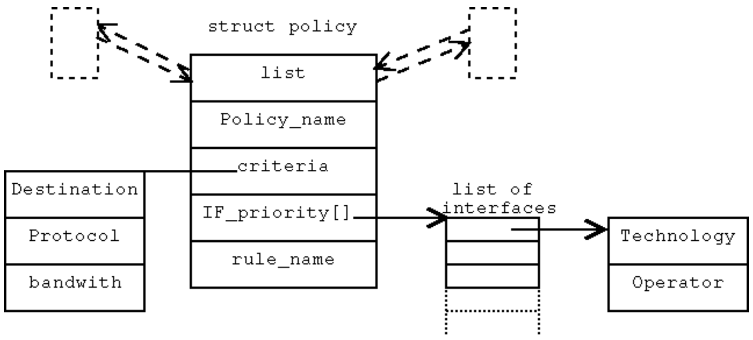


Figure 8. The `policy` structure.

5. THE GUI APPLICATION

From the user's point of view, the GUI consists of four main parts: A list of interfaces, interface details, a ticker and a command menu. In the main view a list of available network interfaces, operators offering those and the status indicators of the respective interfaces (busy, enabled or disabled) are shown. Also the ticker in which common information, error reports and, most importantly, the currently selected policy rule in use are highlighted is shown in the main view. The *figure 9* presents the functionality of the GUI as a UML use-case diagram.

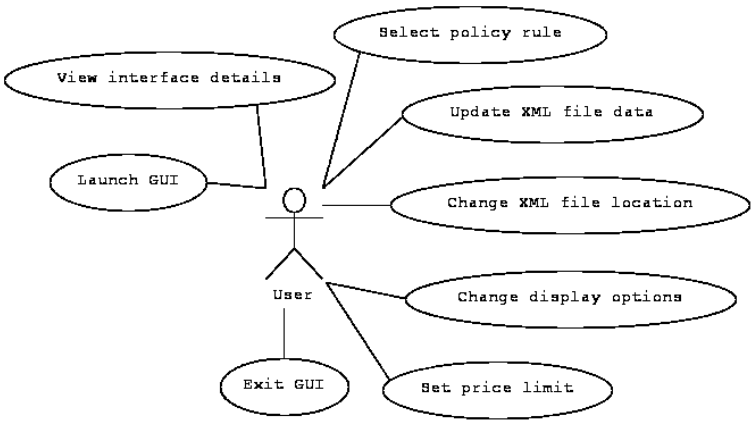
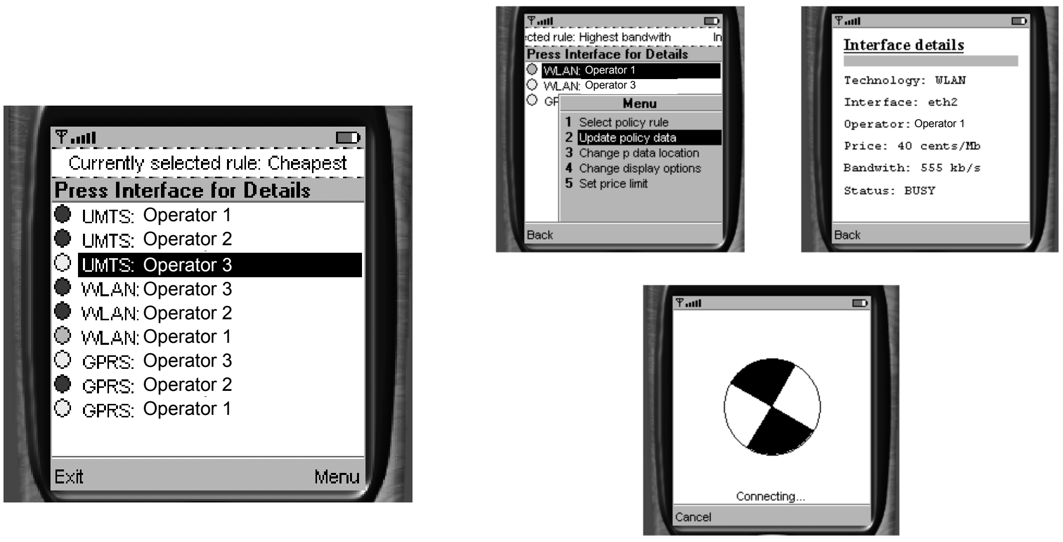


Figure 9. The use-cases.


Currently selected rule: Cheapest
Press Interface for Details
UMTS: Operator 1
UMTS: Operator 2
UMTS: Operator 3
WLAN: Operator 3
WLAN: Operator 2
WLAN: Operator 1
GPRS: Operator 3
GPRS: Operator 2
GPRS: Operator 1
Exit Menu

cted rule: Highest bandwith In
Press Interface for Details
WLAN: Operator 1
WLAN: Operator 3
GF
Menu
1 Select policy rule
2 Update policy data
3 Change p data location
4 Change display options
5 Set price limit
Back

Interface details
Technology: WLAN
Interface: eth2
Operator: Operator 1
Price: 40 cents/Mb
Bandwith: 555 kb/s
Status: BUSY
Back

Connecting...
Cancel

Figure 10. Snapshots from the GUI.

The look-and-feel of the GUI is illustrated in *figure 10*. The GUI has embedded automatic guidance, so that the user can easily operate it. For example, if there isn't any policy rule in use (when the GUI is started, it automatically inquires the API module whether it is already using one), it displays a rule selection menu where the user shall select the most convenient one.

An unsorted list of interfaces is initialized from the policy data XML file, or if it is for some reason unavailable, the initialization is interpreted from the API module's interface state changed messages. Whenever a policy rule is selected, the program sorts the interface list into a priority order according to the selected policy. Then the sorted list is sent to the API module. If the highest priority interface is not available, the API module tries to use the second highest priority on the list and so on. The implementation of the functionality of the main use-cases is illustrated as a UML sequence diagram shown in *figure 11*. No class diagrams are included in this paper for clarity reasons.

The communication with the API module was implemented using sockets. However, due to the lightweight nature of sockets there was a need to define a communication protocol so that the both systems had a mutual understanding. The communication protocol defined the structure and data contents of the messages sent between the GUI and the API module.

In MIDP devices everything is small: The network connection is slow, the processor is slow and memory is scarce. Because of these constrains, MIDP applications should be designed to be as small as possible. This implies also to the XML parsing task. Generally, there are three of kinds XML parsers: model parsers, push parsers and pull parsers. Model parsers read the entire document and create a representation of the document in memory. A push parser reads through the entire document and as it encounters the searched parts of the document, it notifies a listener object. A pull parser reads a little bit of the document at a time, and the program then requests the next piece. A compact model parser *Xparse-J* (Claßen M., 2000) under the GNU Public License was selected to be used in the GUI.

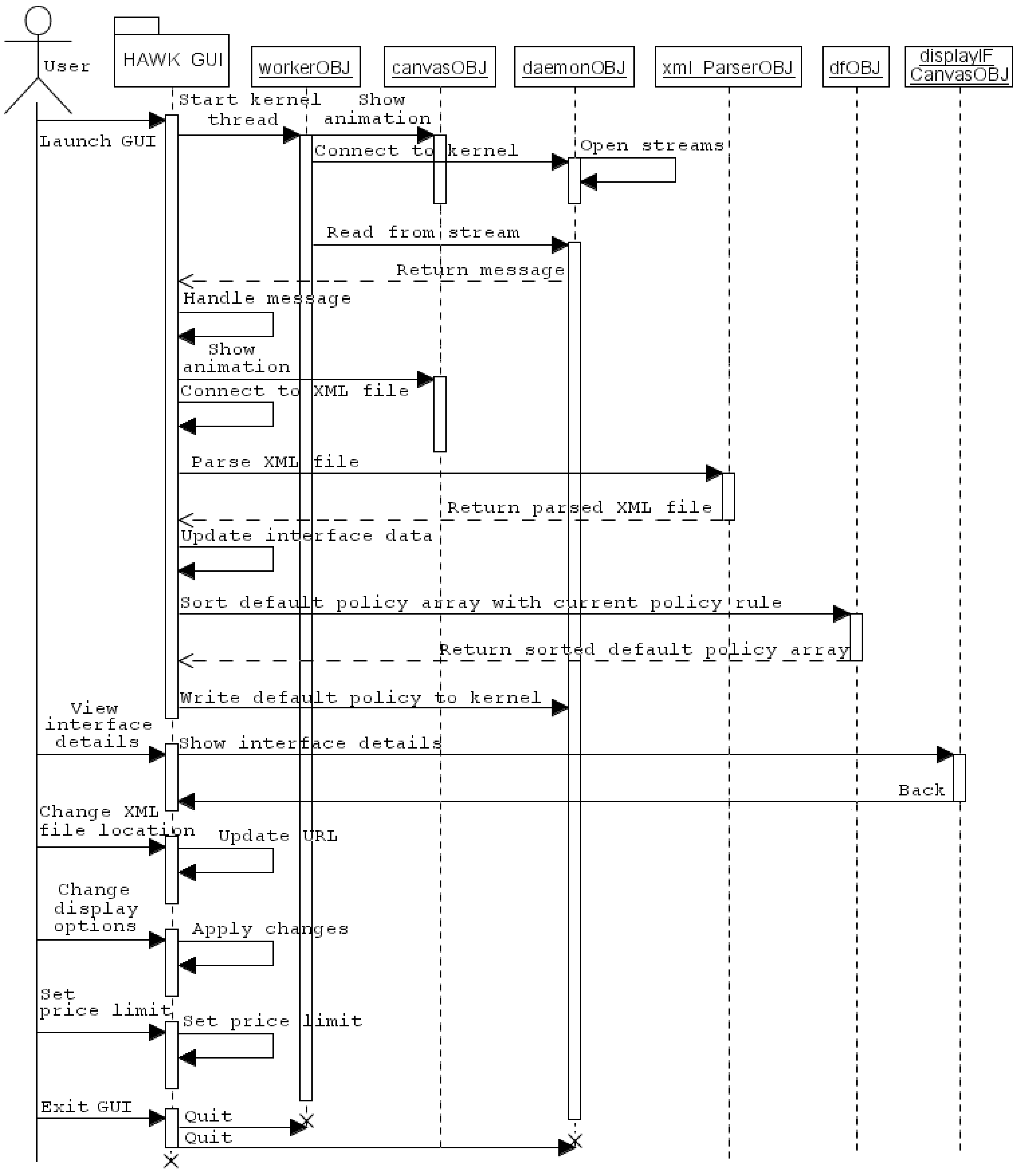


Figure 11. The implementation of the main use-cases.

6. CONCLUSION

In this paper, a prototype implementation integrated with a Mobile IPv6 stack for Linux able the control the heterogeneous network access in a mobile terminal was presented. The prototype project participates in the mobility development, so that while roaming from places to places, the user could retain the best possible network access without interruptions. Through the prototype GUI, the user can set different policy rules defining how the specified interfaces should be sorted into a priority order. The API module then interprets this default policy and performs the interface selection with respect to this policy through the Mobile IPv6 stack. Moreover, if the selected interface is not available or becomes unavailable, the system is able to perform a seamless hand-over to the following available access interface.

Still some issues remain unsolved. For instance, how to store and update the policy data in a real-life environment, and how to fetch reliable information on the interface characteristics, require further research efforts.

ACKNOWLEDGEMENT

The R&D Center of Turku Polytechnic and Oy LM Ericsson Ab are gratefully acknowledged for their financial support.

REFERENCES

Arkko J., Johnson D. and Perkins C. (2003). Mobility support in IPv6. *Internet Draft*, June 30 2003, IETF.

Claßen M. (2000). *Xparse-J XML Parser for Java 1.1*, http://www.webreference.com/xml/tools/.

Deering S. and Hinden, R. (1998a). IP Version 6 Addressing Architecture. *RFC 2373*, IETF.

Deering S. and Hinden, R. (1998b). Internet Protocol, Version 6 (IPv6) Specification. *RFC 2460*, IETF.

Jokikyyny T., Kauppinen T., Ylitalo J., Tuominen A. and Laine J. (2002). *Interface Selection in Simultaneous Multi-access*. Project report (T-106.850), Ericsson Corporate Research & Helsinki University of Technology.

Jokela P., Rinta-aho T., Wall J., Nuorvala V. and Petander H. (2002). *Multiaccess*. Project report (Tik-106.850), Ericsson Corporate Research & Helsinki University of Technology.

Perkins C. (1996). IP Mobility Support, *RFC 2002*, IETF.

Perkins C. (ed.) (2002). IP Mobility Support for IPv4, *RFC 3220*, IETF.

Tuominen A.J. and Petander H. (2001). MILP Mobile IPv6 for Linux in HUT Campus Network Mediapoli. *Proceedings of Ottawa Linux Sumposium*, Canada.

Yavatkar R., Pendalarakis D. and Guerin R. (2000). A Framework for Policy-based Admission Control, *RFC 2753*, IETF.

Yi-an C. (1995). *A Survey White Paper on Mobile IP*.