ROMA: A Middleware Framework for Seamless Handover

Adrian Popescu[1], David Erman[1], Karel de Vogeleer[1], Alexandru Popescu[1,2], and Markus Fiedler[1]

[1] Blekinge Institute of Technology
371 79 Karlskrona, Sweden
adrian.popescu@bth.se
[2] University of Bradford
Bradford, West Yorkshire BD7 1DP, United Kingdom

Abstract. The chapter reports on a new middleware architecture suggested for seamless handover. The middleware is part of an architectural solution suggested by Blekinge Institute of Technology (BTH) for seamless handover, which is implemented at the application layer. This architecture is subject for the PERIMETER STREP and MOBICOME projects, granted by the EU FP7 and EUREKA, respectively. The suggested middleware, called ROMA, represents a software system with two sets of Application Programmer Interface (API), one for application writers and another one for interfacing various overlay and underlay systems. ROMA thus provides a transport-agnostic platform for future Internet applications. The paper provides a short description of the ROMA middleware, with particular focus on API design and address translation.

1 Introduction

Future mobile networks are expected to be all-IP-based heterogeneous networks that allow users to use any system anytime and anywhere. They consist of a layered combination of different access technologies, e.g., UMTS, WLAN, WiMAX, WPAN, connected via an IP-based core network to provide interworking. These networks are expected to provide high usability (anytime, anywhere, any technology), support for multimedia services, and personalization. Technologies such as seamless handover, middleware, multicarrier modulation, smart antenna techniques, OFDM-MIMO techniques, cooperative communication services and local/triangular retransmissions, software-defined radio and cognitive radio are expected to be used.

There are three possibilities to handle movement: at the link layer (L2), network layer (L3) and application layer (L5) in the TCP/IP protocol stack. The complexity of handover is large and demands for solving problems of different nature. Accordingly, a number of standard bodies have been working on handover, e.g., IEEE, 3GPP, 3GPP2, WiMAX, IETF.

The main requirements for handover are in terms of service continuity, context-awareness, security, flexibility, cost, quality, user transparency and a system

D. Kouvatsos (Ed.): Next Generation Internet, LNCS 5233, pp. 784–794, 2011.

architecture that is independent of the access technology. Furthermore, future mobile applications demand for robust fault-tolerance algorithms, able to adapt to imperfect network environments and to provide QoS. This means that elements like middleware are increasingly demanded to provide, among others, support for robust algorithms to guarantee fault-tolerance for mobile applications, to react very adaptively to environment changes and to provide the requested QoS guarantees. The mobile middleware is also requested to integrate different services and resources in a distributed execution environment and supply the users with open and consistent APIs.

Today, there is a quite large number of standardization forums for mobile middleware. Because of this, the situation is quite diffuse, with the consequence of less focus in defining particular middleware solutions for particular situations.

In this context, a new architecture has recently been advanced by Blekinge Institute of Technology (BTH), which is able to provide soft QoS guarantees on top of Peer-to-Peer (P2P) networks. This is an ongoing research, where the main research challenges are on middleware, overlay routing, mobility modeling and prediction, and handover implemented above the transport layer [1].

The rest of the paper is as follows. Section II is about the existent seamless handover solutions. Section III is about middleware requirements. Section IV describes a new architecture suggested for seamless mobility, which is implemented at the application layer. Section V presents the main concepts and results of the suggested middleware as well as some important open issues. Finally, section VI concludes the paper.

2 Seamless Handover

Most of the existent solutions attempt to solve the handover problem at L2 (access and switching) and L3 (IP) with particular consideration given to L4 (transport) [1]. Some of the most important requirements are on seamless handover, efficient network selection, security, flexibility, transparence with reference to access technologies and provision of QoS.

Typically, the handover process involves the following phases: handover initiation; network and resource discovery; network selection; network attachment; configuration (identifier configuration; registration; authentication and authorization; security association; encryption); and media redirection (binding update; media rerouting).

The basic idea of L2/L3 handover is using Link Event Triggers (LET) fired at Media Access Control (MAC) layer, and sent to a handover management functional module such as L3 Mobile IP (MIP) or L3 Fast MIP (FMIP) or IEEE 802.21 Information Server (IS). LET is used to report on changes with regard to L2 or L1 conditions, and to provide indications regarding the status of the radio channel. The purpose of these triggers is to assist IP in handover preparation and execution.

The type of handover (horizontal or vertical) as well as the time needed to perform it can be determined with the help of neighbor information provided by the Base Station (BS) or Access Point (AP) or the IEEE 802.21 Media Independent Handover Function (MIHF) Information Server (IS).

Given the extreme diversity of the access networks, the initial model was focused on developing common standards across IEEE 802 media and defining L2 triggers to make Fast Mobile IP (FMIP) work well. Connected with this, media independent information needs to be defined to enable mobile nodes to effectively detect and select networks. Furthermore, appropriate ways need to be defined to transport the media independent information and the triggers over all 802 media.

In reality, however, the situation is much more challenging. This is because of the extreme diversity existent today with reference to access networks, standard bodies and standards as well as architectural solutions. Other problems are because of the lack of standards for handover interfaces, lack of interoperability between different types of vendor equipment, lack of techniques to measure and assess the performance (including security), incorrect network selection, increasing number of interfaces on devices and the presence of different fast handover mechanisms in IETF, e.g., MIPv4, Fast MIPv6 (FMIPv6), Hierarchical MIPv6 (HMIPv6), Fast Hierarchical MIPv6 (FHMIPv6). Furthermore, implementing make-before-break handovers is very difficult at layers below the application layer.

IETF anticipated L2 solutions in standardized form (in the form of triggers, events, etc), but today the situation is that we have no standards and no media independent form. Other problems are related to the use of L2 predictive trigger mechanisms, which are dependent on L1 and L2 parameters. Altogether, the consequence is in form of complexity and dependence on the limitations of L1, L2 and L3. The existent solutions are generally not yet working properly, which may result in service disruptions.

Today, user mobility across different wireless networks is mainly user centric, thus not allowing operators a reasonable control and management of inherently dynamic users. Furthermore, the traditional TCP/IP protocol stack was not designed for mobility but for fixed computer networks. The responsibility of individual layers is ill-defined with reference to mobility. The main consequence is that problems in lower layers related to mobility may create bigger problems in higher layers. Higher layer mobility schemes are therefore expected to better suit Internet mobility. This kind of solutions opens up for research and development of new architectural solutions for handover based on movement, possibly implemented at L5 in the TCP/IP protocol stack.

3 Middleware Requirements

The main elements of a software architecture for mobile protocol stack are the Operating System (OS), TCP/IP Protocol Stack (TCP/IP), Mobile Middleware (MM) and User Interaction Support (UIS). Different applications obtain services from these entities through Application Programming Interfaces (API). Mobile applications however are distributed and they demand for particular standard protocols that the applications can use in their interactions. A mobile middleware therefore represents an execution environment where a set of generic service

elements like configuration management, service discovery and event notification are defined. Figure 1 shows a typical example of mobile middleware and the generic service elements [2].

Some of the most important requirements for middleware are:

- Provide support for multiple types of mobile platforms, e.g., mobile phones, PDAs, laptops.
- Address the diversity of mobile devices and provide a consistent programming environment across them with high level modeling approaches.
- Provide different types of profile with reference to, e.g., network interface, access network, flow description.
- Provide implementation style (i.e., local, client-server and P2P) invisible to applications.
- Provide support for context-awareness, which means that the mobile applications should be able to adapt to diverse user context changes regarding, e.g., user preferences, terminal and network facilities, user environment and user mobility.
- Provide support for fault-tolerance, which means that the mobile applications should be able to adapt to the particular churn situation existent in the network by using adaptive overlay routing.
- Provide support for lightweight protocols, able to adapt, with minimum of resources, the mobile applications to different domain and environment needs.
- Reduce the gap between the performance of external communications among hosts and internal communication among processes residing on the same machine or within local clusters under common administrative domains.
- Provide security facilities like, e.g., application security, device security, firewall facilities and hosted server policies in the case of using hosted services.
- Provide diverse management facilities like, e.g., backup and restore, mobile system policy control, Over-the-Air (OTA) provisioning, software and hardware inventory management.
- Allow the developers to create applications through an interactive process of selecting the elements of the user interface and the objects they manipulate. Further, local emulation of mobile devices should be offered to developers to test the particular application without installing the particular software on the device.

4 ROMA Architecture

We suggest a new architectural solution for seamless handover, which is implemented at the application layer [1]. Compared to the existent handover solutions, implemented at the link layer and network layer, this solution offers the advantage of less dependence on physical parameters and more flexibility in the design of architectural solutions. By this, the convergence of different technologies is simplified. Furthermore, by using an architecture based on middleware and overlays, we have the possibility to combine the services offered by different

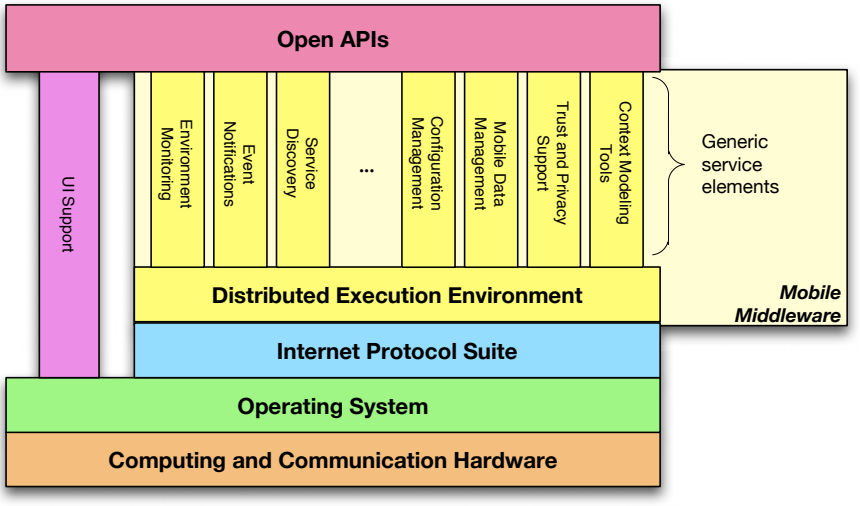

Fig. 1. Mobile middleware and the generic service elements [1]

overlays. This offers the advantage of flexibility in the development of new services and applications. The suggested architecture resembles the Android mobile development platform developed by Google [3], opening thus up for similar architectural solutions developed in the terminal and in the network. By this, new applications and services can be easily designed and developed.

The suggested architectural solution is shown in figure 2. It is based on using a middleware (with a common set of Application Programming Interfaces (APIs)), a number of overlays and a number of underlays. By middleware, we refer to software that bridges and abstracts underlying components of similar functionality and exposes the functionality through a common API. On the other hand, by overlay we refer to any network that implements its own routing or other control mechanisms over another already existing substrate, e.g., TCP/IP, Gnutella. Finally, by underlays we refer to substrates, which are abstracted networks. Bu substrate, we refer to any network (e.g., IP overlay or "normal" IP network) that can perform the packet transportation function between two nodes. Thus, ROMA implements a transport-agnostic enabler for any kind of application.

The underlays can be either structured or unstructured. Structured overlays are networks with specific type of routing geometry decided by the Distributed Hash Table (DHT) algorithm they use. Structured underlays use keys for addressing like, e.g., Chord [4]. In unstructured overlays the topology can be viewed as emergent instead of being decided before hand. Unstructured overlays can use IP addresses or other forms of addressing, e.g., Gnutella, which uses Universal Unique IDs (UUIDs) for addressing.

An important goal of the middleware is to abstract structured and unstructured underlays as well as overlays. This API architecture is used in several projects, relating to overlay routing with soft QoS, and seamless roaming [1].

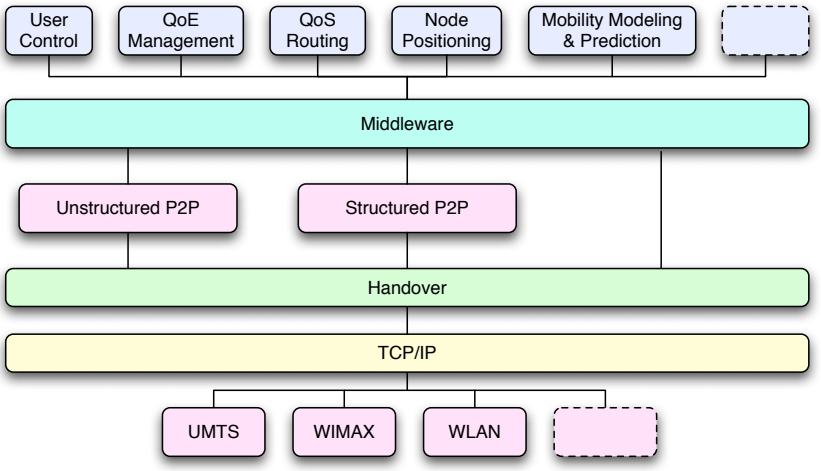

Fig. 2. ROMA architecture

We use the application layer protocol Session Initiation Protocol (SIP) [5] for mobility management. This solution has the advantage of eliminating the need for a mobility stack in mobile nodes and also does not demand for any other mobility elements in the network, beyond SIP servers. Simple IP is used in this case together with a SIP protocol stack. The drawback however is because the existing client frameworks do not accommodate IETF SIP [5] and 3GPP SIP [6] within the same framework. The consequence is that today one needs two different sets of client frameworks on the mobile, one for the mobile domain (e.g., UMTS) and the other one for the fixed domain (e.g., fixed broadband access in combination with WLAN).

BTH has also co-developed an interesting solution for vertical handover, called the Network Selection Box (NSB) [7,12]. NSB encapsulates the raw packet in a UDP datagram and sends it over a real network. A tunneling concept is used to send the packets over the interfaces encapsulated in UDP. The NSB can today be used for the transport over WLAN, UMTS and GPRS. While the NSB does solve the issue of inter-technology handovers, ROMA provides a comprehensive support structure for interoperation of both applications and various transport substrates.

5 ROMA Middleware

The main goal of the project is to develop a testbed to facilitate the development, testing, evaluation and performance analysis of different solutions for user-centric mobility, while requiring minimal changes to the applications using the platform. In other words, we implement a software system with two sets of APIs, one for application writers and another one for interfacing various overlay and underlay systems.

Current overlay implementations are built with incompatible language specific frameworks on top of the low level networking abstractions, e.g., YOID, i3, JXTA [4,8]. This complicates the design of overlays and their comparison as well as the integration of different overlays. We therefore suggest a middleware based on the Key-Based Routing (KBR) layer of the common API framework suggested in [8]. By doing so, independent development of overlay protocols, services and applications is facilitated.

The middleware is intended to work on top of both structured and unstructured underlays. Structured underlays can be used to construct services such as Distributed Hash Tables (DHT), scalable group multicast/anycast and decentralized object location. The advantage is that they support highly scalable, resilient, distributed applications like cooperative content distribution and messaging. Unstructured overlays do not have such facilities, but they tend to have less overhead in handling churn and keyword searches [9].

By using a common API, we can develop applications by using combinations of arbitrary overlays and underlays. This facility allows us to design a testbed where we can investigate interoperability issues and performance of different combinations of protocols. This also allows us to have overlays that export APIs that other overlays can use. For instance, we can have the "Quality of Experience (QoE) Management" export an API that can be used by the "Quality of Service (QoS) Routing" overlay and "Handover" underlay.

5.1 API Design

The ROMA API is based on the KBR layer of the common API framework suggested in [8]. Our approach differs in one major aspect. In [8] it is assumed that the KBR API runs on top of a structured underlay only. In our case, the ROMA API operates on top of both structured and unstructured underlays. This means that both structured and unstructured overlays are able to use the ROMA API.

This feature allows for the development of applications using combinations of arbitrary, ROMA-based, overlays and underlays. This allows us to design testbeds that investigate the interoperability and performance of various protocol combinations. For instance, given a new congestion control algorithm, one can test how it performs when running on top of Chord or Content Addressable Network (CAN) or BitTorrent or IP, respectively.

Furthermore, since the ROMA API is an enabler for seamless handover, the API is designed to provide the ability to run several overlays and several underlays simultaneously. The advantage is that one can have some overlays export APIs that other overlays can use, and this is valid for underlays as well. For example, one can have a measurement overlay exporting an API that can be used by a resource reservation overlay to reserve a path satisfying specific constraints. The resource reservation overlay can in turn export an API, which is used by a video conference application that requires some QoS guarantees between endpoints. This means that basic services can be chained or combined into one or

more complex services, which is an important part of the dynamic composability expected of future Internet services.

The API core is the KBR abstract base class. The KBR class specifies the API functions in terms of abstract class methods. The API functions are the same as those presented in [8], and we refer to this for the semantics of each function. It is important to mention that the KBR class does not implement these functions, but it expects derived classes to implement them.

Given the software implementation of an API underlay, say CAN, one may need to adjust it so it can easily provide the functionality required be the ROMA API. This implies wrapping the CAN implementation into a class that translates between function calls supported by the ROMA API and functions supported by the CAN API. The wrapper class inherits (in the OOP-sense) the KBR class and it is forced to provide all API functions. We call these wrapper classes shim layers and expect to have one shim layer for each underlay we wish to plug into the API.

An overlay can either extend (OOP inheritance) one shim layer or act as a container for several shim layers, depending on the intended application. By exporting the API of an overlay we mean merely providing header files that other overlay can use to instantiate objects of the exporting overlay and obtaining services from it. Although one can discover such APIs at runtime, through introspection for instance, this is not currently in the scope of our project.

5.2 Architecture

Asynchronous system calls are used within the middleware. This is achieved by means of the *boost* framework [10]. In particular the *Asio* package is utilized, which provides us with portable networking solutions, including sockets, timers, hostname resolution and socket iostreams. The middleware does not block upon I/O system calls, e.g., sockets writing and reading, as a consequence of the asynchronous properties of the boost's *Asio* package. Hence the middleware does not show any CPU hogging behaviour and can operate in a conventional fashion with slow I/O streams.

Table 1. Overview of the Middleware's API

Function	Description
`init()`	Initializes the application.
`run()`	Starts the execution of the application.
`route()`	Routes a message towards a destination.
`deliver()`	Upcall from the middleware to deliver a message.
`forward()`	Upcall from the middleware to forward a message.
`addUnderlay()`	Adds an middleware to the application.
`addOverlay()`	Adds an application to the middleware.
`lookup()`	Checks whether a specific service is bound to the middleware (service discovery).

The middleware consists of a shim layer, `kbr`, that is inherited by the classes wanting to utilize the functionalities of the middleware. The shim layer, an Application Programming Interface (API), provides basic functions by which the middleware functionalities can be exploited. Table 1 elaborates the API accessible for applications.

Messages originating from an application are exchanged between another middleware entity and then forwarded to the destination application. Applications running on top of the middleware are identified by *keys*. The *Keys* can be used as identifiers for applications but also to refer to other entities in the middleware stack such as the middleware core itself or to identify files. These *keys* are formated as standardized UUIDs by the ITU-T [11], 128 bit or 16 bytes long.

5.3 Address Translation

The most important tasks of the ROMA middleware are to provide an addressing and routing substrate, pseudo-asynchronous event management and non-blocking TCP connection management.

The KBR API defines a 160 bit addressing scheme and a small set of functionality for routing messages. It fits the purpose of the ROMA architecture well. If, during the development of the ROMA architecture, the KBR API is found to be lacking, then this will be extended or modified to suit the purposes of ROMA.

ROMA layers address nodes by using keys. However, different overlays (e.g., Chord, CAN, Kademlia) use different keys or key formats. This may demand that a layer knows how to convert its key format to the key format of any other layer it wants to talk to. This means that whenever a new ROMA layer is designed one must extend all other layers to understand the new key format. This is clearly undesirable. To avoid this, we introduce a common key format, defined in [8]. Whenever a layer needs to communicate an address to another layer (above or below), it converts its own key to the common key format. If layers follow this rule, then whenever a new layer is added, the particular layer needs only to know how to convert from its own key format to the common key format and viceversa. By this, none of the other layers need to be changed.

A common key is defined as a 160-bit string [8]. Specific keys can be strings that are longer or shorter. Roughly speaking, when the specific key is shorter than the common key, the unused high bits of the common key are masked. In the case the specific key is longer than the common key, then the specific key is truncated to fit into the common key. When a specific key is mapped by truncation to an existing (used) common key, then the next common key is chosen. If the common key is used as well, then the process is repeated until an unused common key is found. We assume that it is unlikely that as many as 2^{160} keys can be active simultaneously. However, the truncation may mean that no guarantees can be provided that the common keys are unique throughout the network. Therefore, we require that keys are unique only within the scope of a ROMA stack on one host. This is not necessarily a problem since either IP addresses or a DHT with specific keys can provide uniqueness.

The general rule is that all common keys are created by the bottom layer (closest to the physical layer). Layers in between learn about keys when their *getKey()*, *route()*, *forward()* and *deliver()* functions are being called. The last three functions have the usual semantic as described in the KBR API [8]. The *underlay.getKey(Socket)* function is called when a layer needs to obtain a common key for the IP address in the *Socket* variable. The function is called by each underlay until it reaches the bottom layer. The bottom layer checks if the socket is in use, in which case it returns the corresponding common key. If the socket is not in use, a new common key is mapped to the IP address and returned to the layer above. Each layer can then create a specific key corresponding to the common key, before returning from the function call. This procedure guarantees the uniqueness of the common key across the stack.

5.4 Situation Today

The basic functions of the middleware have today been implemented. The middleware is able to transmit and receive message in an asynchronous manner. The middleware maintains a set of connections used by the applications that are linked to the middleware. Data streams are automatically relayed by the middleware with a smart switch between application and connection and vice versa.

The current ROMA implementation is in the C++ language, with in-house developed wrapper classes around the glib library and the *epoll()* linux system call for event handling. This makes the middleware tied to the linux kernel. We have therefore changed this to use the Boost asynchronous IO library instead, to make the middleware more platform-agnostic [10].

5.5 Open Issues

An important design assumption for the ROMA middleware is to have access to more than one network interface with IP functionality. In addition to this, each interface may correspond to more than one underlay, by using compositions of underlay shims. This means that, for an overlay making use of the ROMA API, data can be received and transmitted on several underlays. The consequence is that the middleware may be required to perform multiplexing/demultiplexing of the dataflows through the middleware.

In the current incarnation of the middleware, each application linking to the middleware gets its own ROMA instance. This means that flow multiplexing/demultiplexing does not need to be performed between applications. However, this also means that the functionality is duplicated for each instance as well as that addresses are not guaranteed to be unique between different ROMA applications.

A network interface switch running in the kernel will therefore be used to switch between technologies and it is currently under development. The middleware is able to communicate with this in-kernel switch and can retrieve vital information for decision making and handovers.

6 Conclusions

The paper has reported on a new middleware architecture suggested for seamless handover. The middleware is part of an architectural solution suggested by Blekinge Institute of Technology for seamless handover, which is implemented at the application layer. The paper has particularly focused on the main design elements, namely API design and address translation.

In the future, we will further implement the suggested middleware as well as test it in real mobile environments. The expected results will be used in the EU FP7 projects PERIMETER and MOBICOME, in which BTH is actively participating.

References

1. Popescu, A., Ilie, D., Erman, D., Fiedler, M.: An Application Layer Architecture for Seamless Roaming. In: 6th International Conference on Wireless On-demand Network Systems and Services, Snowbird, Utah, USA (February 2009)
2. Raatikainen, K.: Recent Developments in Middleware Standardization for Mobile Computing, Nokia Research Center, Finland
3. Android, An Open Headset Alliance Project, `http://code.google.com/android/`
4. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: ACM SIGCOMM 2001, San Diego, USA (August 2001)
5. Rosenberg, G., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol. IETF RFC 3261, `http://www.ietf.org`
6. ETSI/3GPP, Universal Mobile Telecommunication System (UMTS): Signaling Interworking Between the 3GPP Profile of the Session Initiation Protocol (SIP) and non-3GPP SIP Usage, 3GPP TR 29.962 version 6.1.0 Release 6, `http://www.3gpp.org/ftp/specs/html-info/29962.htm`
7. Isaksson, L.: Seamless Communications Seamless Handover Between Wireless and Cellular Networks with Focus on Always Best Connected. PhD thesis, BTH, Karlskrona, Sweden (March 2007)
8. Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a Common API for Structured Peer-to-Peer Overlays. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735. Springer, Heidelberg (2003)
9. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making Gnutella-Like P2P Systems Scalable. In: ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications, Karlsruhe, Germany (2003)
10. Boost C++ Libraries (2009), `http://www.boost.org`
11. Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components, ITU-T Rec. X.667 | ISO/IEC 9834-8
12. Chevul, S., Isaksson, L., Fiedler, M., Lindberg, P.: Network selection box: An implementation of seamless communication. In: García-Vidal, J., Cerdà-Alabern, L. (eds.) Euro-NGI 2007. LNCS, vol. 4396, pp. 171–185. Springer, Heidelberg (2007)