國立交通大學

資訊科學與工程研究所

碩士論文

SIP 終端設備移動性在 IPv4 和 IPv6 網路之間的應用

SIP Terminal Mobility between IPv4 and IPv6 Networks

研 究 生：葉哲華

指導教授：林一平　教授

吳坤熹　教授

中 華 民 國 九 十 五 年 七 月

SIP 終端設備移動性在 IPv4 和 IPv6 網路之間的應用

SIP Terminal Mobility between IPv4 and IPv6 Networks

研 究 生：葉哲華　　Student：Che-Hua Yeh

指導教授：林一平　　Advisor：Yi-Bing Lin

吳坤熹　　Quincy Wu

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

SIP 終端設備移動性在 IPv4 和 IPv6 網路之間的應用

學生：葉哲華　　指導教授：林一平 教授
吳坤熹 教授

國立交通大學資訊科學與工程學系（研究所）碩士班

摘　　要

Session Initiation Protocol (SIP)支援應用程式層的移動性。在這篇碩士論文中，我們將會介紹 SIP 終端設備移動性(SIP Terminal Mobility)在 IPv4 和 IPv6 網路之間的應用以及其實作。我們在兩個 SIP 通訊軟體中實作了 SIP 終端設備移動性，並且在純 IPv4 網路、純 IPv6 網路以及 IPv4 和 IPv6 之間作其效率測試。實驗結果證明 SIP 終端設備移動性是可以充分有效應用在 IPv4 和 IPv6 網路之中。

SIP Terminal Mobility between IPv4 and IPv6 Networks

Student：Che-Hua Yeh

Advisors：Prof. Yi-Bing Lin
Prof. Quincy Wu

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Session Initiation Protocol (SIP) supports application layer mobility. In this thesis, we will investigate SIP-based mobility across IPv4 and IPv6 networks. The architecture designs based on two SIP libraries for SIP terminal mobility are described, and the performance of SIP User Agents are measured from empirical experiments in IPv4-only, IPv6-only and IPv4/IPv6 networks. The results show that SIP terminal mobility across IPv4 and IPv6 networks can be efficiently supported.

Acknowledgment

This thesis is the result of two years of work whereby I have been accompanied and supported by many people. It is a pleasant that I have now the opportunity to express my gratitude for all of them.

I would first like to thank Prof. Yi-Bing Lin (林一平) and Prof. Quincy Wu (吳坤熹) for their guidance, insight and support throughout the preparation of this thesis.

I would like to thank Dr. Whai-En Chen (陳懷恩), Shiang-Ming Huang (黃祥鳴), and Chia-Yung Su (蘇家永) for the helpful discussions about the design of SIP terminal mobility.

I would also like to thank my master committee members, Prof. Phone Lin (林風) and Prof. Shun-Ren Yang (楊舜仁) for their helpful comments and suggestions.

My colleagues of Laboratory 117 all gave me the feeling of being at home at work. Many thanks for being your colleage.

Finally, I would like to thank my families and all my friends who ispire me or make me feel confidence to accomplish this thesis.

Che-Hua Yeh

July 2006

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Nowadays many users have two independent communication systems on their desktops – a telephony network and a computer network. With an effort to converge these two traditionally independent services, many service providers begin promoting the Voice over Internet Protocol (VoIP) technology, which enables us to make telephone calls using a broadband Internet connection instead of a regular telephone line. In VoIP industry, Session Initiation Protocol (SIP) developed by Internet Engineering Task Force (IETF) is emerging and widely adopted as the signaling protocol of VoIP. Meanwhile, as VoIP technology is applied to both fixed and mobile users, it becomes an important issue to allow computers and communication devices to maintain connection in a mobile environment. In this thesis, we will investigate SIP-based mobility across IPv4 and IPv6 networks. The architecture design on the protocol stack for SIP terminal mobility is described, and the performance of SIP User Agents are measured from empirical experiments.

1.1 SIP

Session Initiation Protocol (SIP) is an application-layer signaling protocol for Internet multimedia session establishment, modification, and termination [1]. All SIP messages are either requests or responses. Six basic methods of SIP are defined in [1]: INVITE, ACK, BYE, CANCEL, OPTIONS, and REGISTER. The INVITE method is used to establish a call. The

ACK method is used to confirm that the client has received a final response to an INVITE request. The BYE method is used to terminate a call. The CANCEL method is used to cancel a pending SIP request before the final response is received. The OPTIONS method is used to query the capabilities of SIP servers. The REGISTER method is used to register IP address information to a SIP Registrar. Meanwhile, SIP has the following response codes: 1xx for informational responses, 2xx for successful responses, 3xx for redirection responses, 4xx for client failure responses, 5xx for server failure responses, and 6xx for global failure responses.

Figure 1 illustrates the SIP registration and call setup procedure in Steps 1-8. Suppose that a Mobile Host (MH) locates in Network A, and a SIP multimedia session is established between the MH and the Correspondent Host (CH), where both the MH and the CH are SIP User Agents (UAs). The SIP Proxy, which also acts as a SIP Registrar here, can accept the REGISTER requests, record the IP addresses of SIP UAs, and forward SIP messages to registered IP addresses of destination SIP UAs. In the figure, the host names of the MH, the CH and the SIP Proxy are "pc1.club.tw" (IPv4 address: 1.1.1.1), "pc2.club.tw" (IPv4 address: 2.2.2.2), and "sip.club.tw" (IPv4 address: 3.3.3.3), respectively. The details of Steps 1-8 are described as follows.

Steps 1 and 2. The CH registers its IP address to the SIP Proxy with a REGISTER request, and the SIP Proxy replies a 200 OK response to indicate the registration is successful.

Step 3. The MH sends an INVITE request via Network A to the SIP Proxy. Figure 2 (a) shows the INVITE message format with the Session Description Protocol (SDP) content. The request URI in the message is "sip:CH@sip.club.tw" (Figure 2 (1)). The username part of the request URI ("CH") is used by the SIP Proxy to retrieve the CH's

registered IP address.

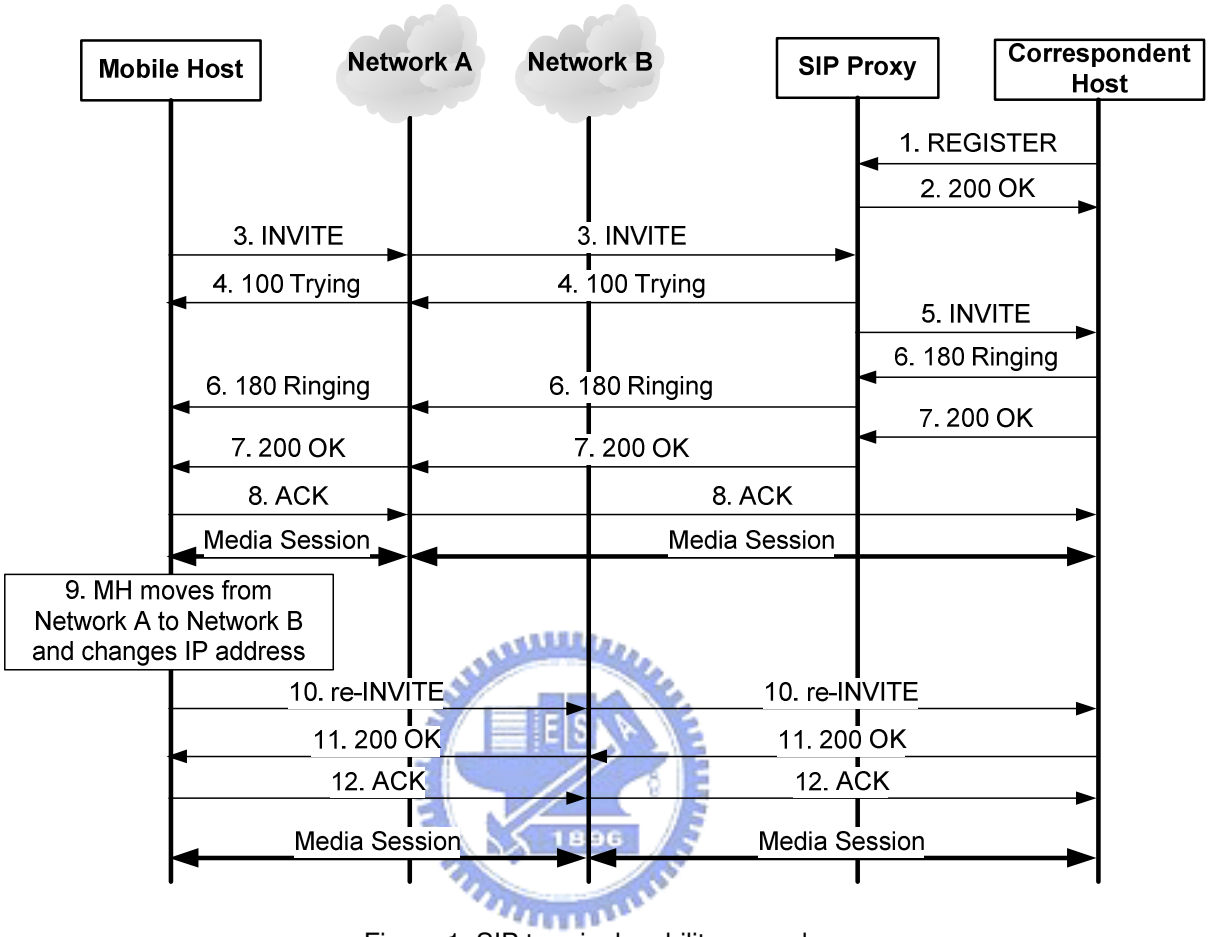


Figure 1. SIP terminal mobility procedure

Steps 4-6. The SIP Proxy sends a 100 Trying response to notify the MH that the SIP Proxy has received the INVITE request and is processing the request. The SIP Proxy then forwards the INVITE request to the CH. After receiving the INVITE request, the CH alerts the called party and sends a 180 Ringing response to the MH through the SIP Proxy.

Steps 7 and 8. The CH answers the call and sends a 200 OK response to the MH through the SIP Proxy indicating that the call is accepted. The IP address of the CH is sent back to the MH through the 200 OK message so that a media session is

established between the MH and with CH. In this way, during the signaling process both the MH and the CH learns the IP address of each other, so the MH can contact the CH directly in the subsequent SIP messages and media sessions. The MH sends an ACK message to the CH to confirm the media session establishment.

After Step 8, the media session is established between the MH and the CH without involving the SIP Proxy, and the SIP basic call setup procedure is finished. Steps 9-12 in Figure 1 are the SIP terminal mobility procedure, which will be described in next section.

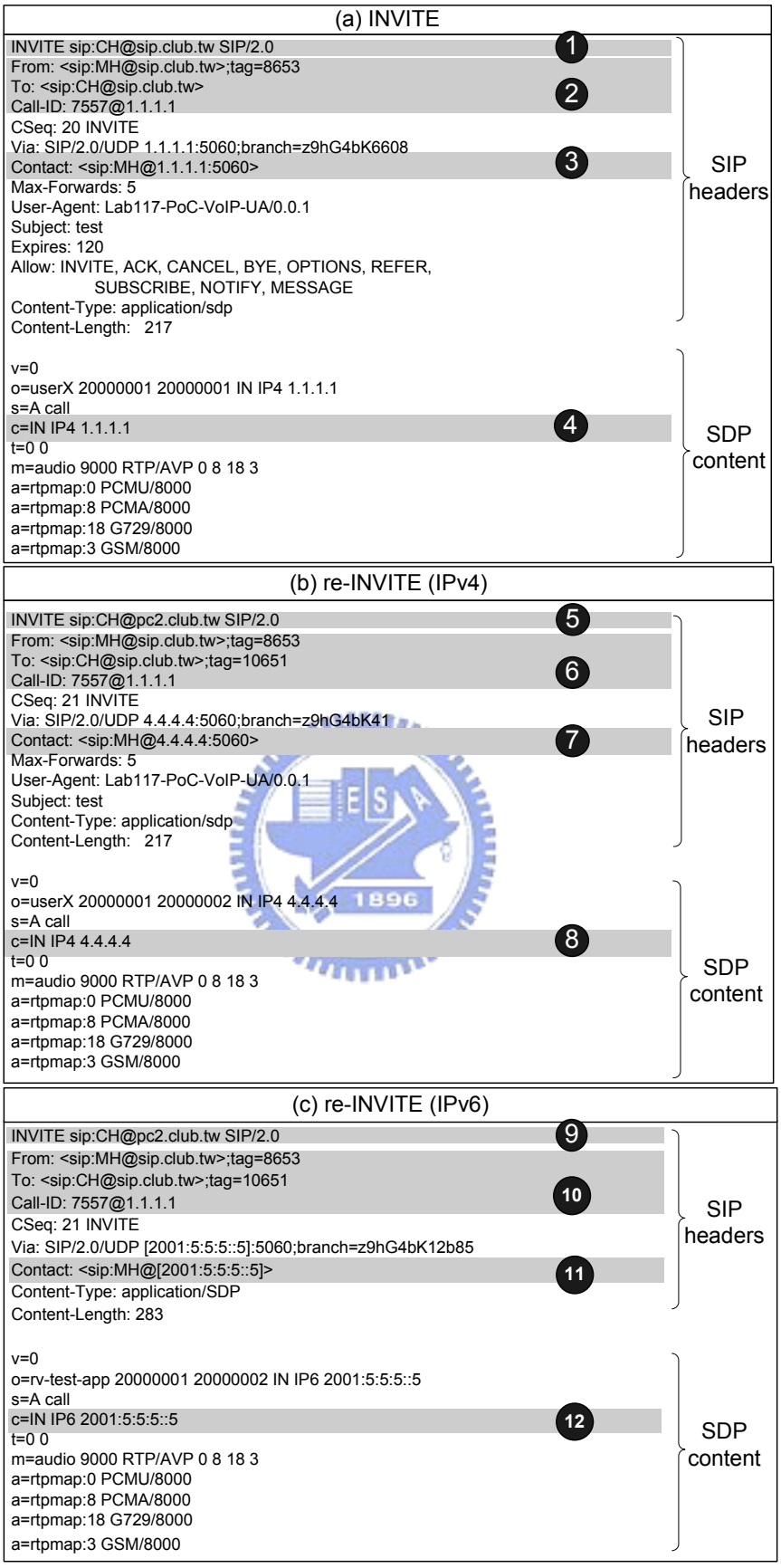


Figure 2. SIP INVITE message and re-INVITE message

1.2 SIP Terminal Mobility

SIP supports four types of mobility, i.e. terminal mobility, session mobility, personal mobility, and service mobility [2]. Terminal mobility is the capability to keep a session alive after the terminal device moves to a different IP subnet. Session mobility is the capability to maintain a session while the user is changing the terminal device. Personal mobility allows a user to become reachable at different terminal devices by the same logical address. Service mobility is the capability to access the user's services (e.g. address book, speed dialing, buddy lists) while the user is moving or changing devices and network service providers.

Figure 1 illustrates the SIP terminal mobility procedure in IPv4/IPv6 networks. Steps 1-8 is the basic SIP registration and call setup procedure described in the previous section. Steps 9-12 are the SIP terminal mobility procedure and the details are described as follows.

Step 9. During the SIP multimedia session, the MH moves from Network A (IPv4) to another Network B (IPv4). The MH acquires a new IP address 4.4.4.4 through, for example, Dynamic Host Configuration Protocol (DHCP) in Network B.

Step 10. The MH sends a re-INVITE request to the CH. The format of the re-INVITE message is shown in Figure 2 (b). In this message, the request URI, "sip: CH@pc2.club.tw" (Figure 2 (5)), is the contact address of the CH which is known by the MH at Step 4. Note that the header fields "From" (indicating the calling party), "To" (indicating the called party), and "Call-ID" (the call session identifier) must be the same as those in the INVITE message (see Figure 2 (2) and Figure 2 (6)). The header field "Contact" is updated to the MH's new IP address (from 1.1.1.1 to 4.4.4.4;

see Figure 2 (3) and Figure 2 (7)). This new address will be used by the CH to contact the MH. The SDP content is also updated in the re-INVITE message. Specifically, the connection address field ("c=") is changed to the MH's new IP address (from 1.1.1.1 to 4.4.4.4; see Figure 2 (4) and Figure 2 (8)).

In the above procedure, the re-INVITE message notifies the calling party to change media transmission parameters. Its format is exactly the same as the INVITE message. Steps 11-12 are the same as Steps 7-8 except that the messages pass through Network B instead of Network A.

1.3 SIP Terminal Mobility between IPv4 and IPv6 Networks

In the previous studies (see [2] and [3]), SIP terminal mobility was discussed in IPv4-only or IPv6-only networks. This thesis elaborates SIP terminal mobility between IPv4 and IPv6 networks. The performance results measured from empirical experiments will also be investigated.

To support SIP terminal mobility between IPv4 and IPv6 networks, the MH must be equipped with the dual-stack SIP UA which supports both IPv4 and IPv6 SIP protocols. On the other hand, the CH can be IPv4-only, IPv6-only, or dual-stack.

Suppose that the MH connects a SIP session with a dual-stack CH through Network A (IPv4). Standard SIP call setup procedure is executed as described in Section 1.1. The MH activates the SIP terminal mobility procedure after moving to Network B (IPv6). In this

network, the MH is assigned an IPv6 address through IPv6 address autoconfiguration [4]. Since the CH is able to receive both IPv4 and IPv6 SIP messages, the CH receives the re-INVITE message sent from the MH through Network B (IPv6). The message flow is exactly the same as that shown in Figure 1 except that messages in Steps 10-12 are transported by IPv6. Figure 2 (c) shows the re-INVITE message in IPv6. The request SIP URI in the re-INVITE message is "sip: CH@pc2.club.tw" (see Figure 2 (9)) which is the contact address of the CH. Because the CH is located in an IPv4/IPv6 dual-stack network, an IPv4 address and an IPv6 address are assigned to the host name, "pc2.club.tw". Therefore the MH can query the Domain Name Service (DNS) server to obtain the CH's IPv6 address through this host name and the re-INVITE can be sent to the CH correctly through IPv6 network (i.e., Network B). The connection address (see Figure 2 (12)) of the re-INVITE message is now an IPv6 address. The header fields "From", "To", and "Call-ID" in the re-INVITE message (see Figure 2 (10)) must be the same as those in the INVITE message (see Figure 2 (2)) as described in the previous section. The header field "Contact" is updated to the MH's new IPv6 address (see Figure 2 (3) and Figure 2 (11)).

Consider another scenario where the CH is located in an IPv4-only network. The simple SIP terminal mobility procedure described above will not work in that case. After the MH moves from Network A (IPv4) to Network B (IPv6) and activates the SIP terminal mobility procedure, the re-INVITE message will not reach the CH because IPv4 routing system is independent from IPv6 routing system. To resolve this issue, SIPv6 Translator was suggested [5]. The SIPv6 Translator consists of two components, NAT-PT (Network Address Translation and Protocol Translation) for IP layer address translation [6] and SIP ALG (Application Level Gateway) for SIP layer address translation [7]. The SIPv6 Translator supports communication between an IPv4 SIP UA and an IPv6 SIP UA. Figure 3 shows how the SIPv6 Translator is configured in this scenario. After the MH moves to Network B (IPv6), the MH activates SIP

terminal mobility. Since the CH is in an IPv4 network, the MH sends the re-INVITE request to the SIPv6 Translator. The SIPv6 Translator replaces the IPv6 address information in the re-INVITE message with its own IPv4 address, and the re-INVITE message is sent to the CH. After the CH accepts the re-INVITE request, the 200 OK response is sent back to the SIPv6 Translator. Similarly, the address information in the 200 OK response is replaced by the IPv6 address of the SIPv6 Translator. After the MH sends the ACK message, the MH and the CH send RTP packets to the SIPv6 Translator, and the SIPv6 Translator forwards the packets to the CH and the MH respectively. Thus, the RTP session between MH and CH is re-established through the SIPv6 Translator.

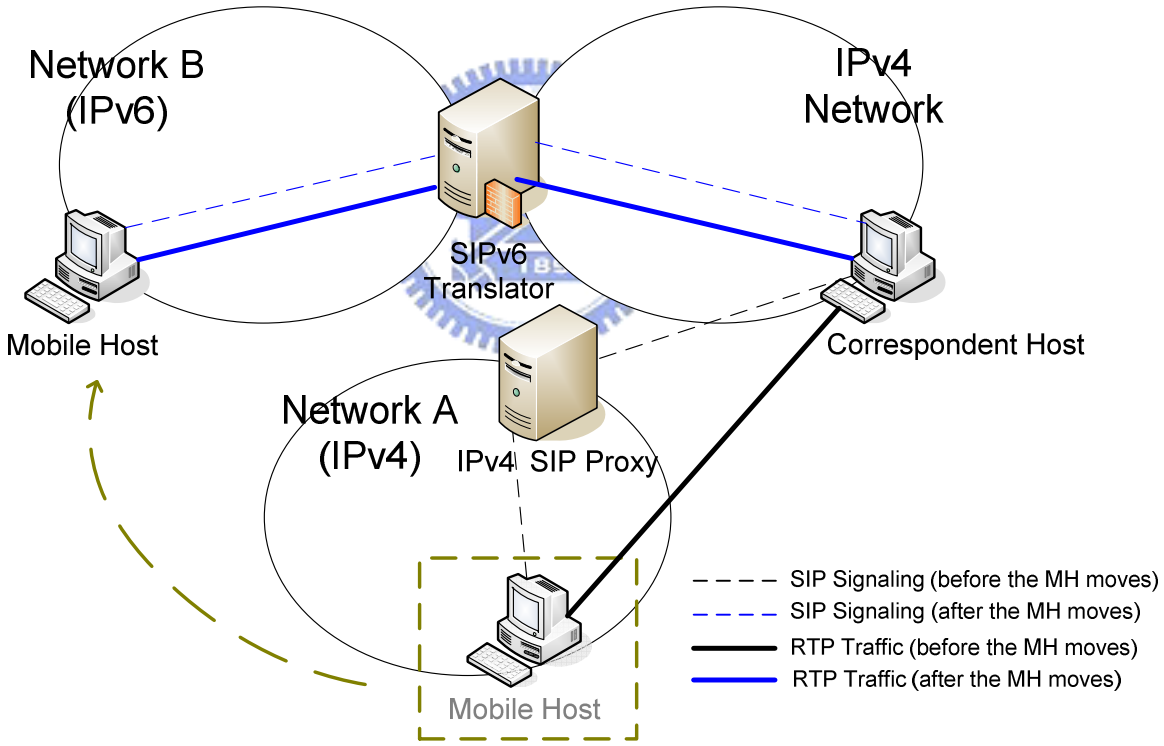


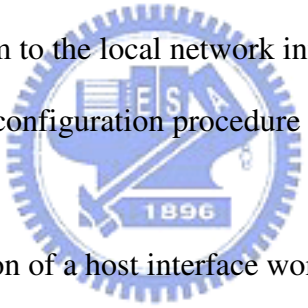
Figure 3. Network configuration for SIPv6 Translator

Details of SIPv6 translator is out of the scope of this thesis and can be found in [5]. In this thesis, we will focus on dual-stack CH such that SIP terminal mobility between IPv4 and IPv6 networks can be supported by modifying the SIP UA of the MH only. In this scenario, no

network node (such as SIPv6 translator) is required, and the SIP UA of the CH needs not be modified.

1.4 DAD in IPv6 Networks

The IPv6 stateless autoconfiguration feature, which enables a host to automatically configure its interfaces with new IPv6 addresses [4]. The automatic configuration of a host interface is performed without the use of a server, such as a DHCP server, or manual configuration. During the procedure of the automatic configuration, Duplicate Address Detection (DAD) is an algorithm to ensure that all configured addresses will be unique on a given link before assigning them to the local network interface. In this section, we will describe the IPv6 stateless autoconfiguration procedure and how DAD is executed.



The automatic configuration of a host interface works in the following way: when a host moves to an IPv6 network, the host will receive the Router Advertisement (RA). The RA is either periodically sent by the IPv6 router, or be delivered to an IPv6 host as the IPv6 router received a Router Solicitation (RS) from the IPv6 host. After receiving the RA the host is allowed to generate its own IPv6 addresses (referred to as the *tentative address*) according to the prefix information contained in the RA. In [4] an algorithm is described on how to automatically compute the IPv6 address of an IPv6 host. For example, if the Ethernet card of a host has the MAC address 00:11:5B:3A:71:E8, then when it moves to a subnet with prefix 2001:238::/64, its 48-bit MAC address will be converted to IEEE EUI-64 format [8], which is a 64-bit interface identifier 0211:5BFF:FE3A:71E8. The IPv6 address 2001:238::0211:5BFF:FE3A:71E8 is (temporarily) assigned to the host. Since the tentative address may be duplicated, the DAD algorithm is executed to ensure there is no address

confliction before the host can send packets via this IPv6 address. In the DAD process, after constructing the IPv6 address according to the received IPv6 subnet prefix, the host broadcasts a Neighbor Solicitation (NS) message on the local link. If no response is received within a pre-determined period, this address will be assigned to the network interface. The average DAD delay ranges from 1 to 2 seconds if there is no address confliction [4].

Chapter 2
Two SIP Terminal Mobility Implementations

To implement SIP terminal mobility in a SIP UA, an extra component is added into the SIP UA. The component must support the following functions: (1) It can detect the modification of local IP addresses, retrieve local IP addresses, select new local IP address, and (2) instruct the SIP UA to send the re-INVITE message. In our implementations, the Winsock API and IP Helper API provided by Microsoft platform are utilized to implement these functions.

There are several choices of SIP libraries for SIP programmers, such as eXtended osip (eXosip [9]), sipX [10] , RADVISION SIP Toolkit [11], Signalware SIP [12], jSIP [13], and JAIN-SIP [14]. Among these SIP libraries, eXosip, sipX, RADVISION SIP Toolkit, and Signalware SIP are written in C and C++, while jSIP and JAIN-SIP are written in Java. Moreover, RADVISION SIP Toolkit and Signalware SIP are commercial products, while the others are free softwares. All these SIP libraries support both Unix-based and Win32 platforms, so it is convenient for programmers to develop their applications on various platforms. In our study, we implemented SIP terminal mobility based on eXosip and RADVISION SIP Toolkit. In the eXosip implementation, SIP terminal mobility is implemented for homogeneous IPv4-only or IPv6-only networks. In the RADVISION

implementation, SIP terminal mobility is implemented for heterogeneous IPv4/IPv6 networks.

2.1 eXosip Implementation

The eXosip library is an open-sourced library based on the GNU osip library [15], which provides software developers an easy and powerful interface to implement SIP applications. The eXosip library provides a high-level API for using the osip library to support SIP multimedia session establishment. In the SIP UA designed and implemented in National Chiao Tung University (NCTU) [16], eXosip is utilized to implement the SIP protocol stack. However, due to the limitation of current version of eXosip, at any single moment eXosip can only bind either an IPv4 address or an IPv6 address to send/receive SIP messages. Therefore the NCTU SIP UA can only support SIP terminal mobility for homogeneous IPv4-only or IPv6-only environments.

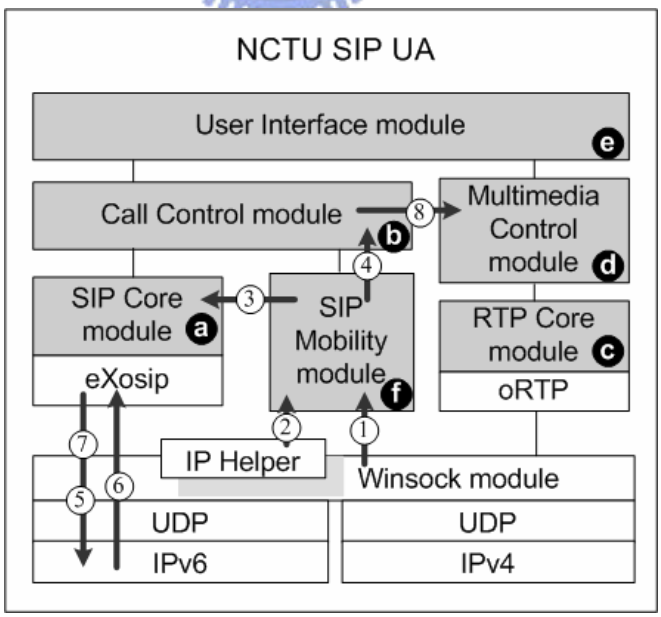


Figure 4. NCTU SIP UA architecture

Figure 4 illustrates the software architecture of the NCTU SIP UA. In this figure, gray boxes represent the SIP UA components we implemented, and white boxes are open-sourced libraries, Winsock and IP Helper APIs, and network protocols provided by the operating system.

In the SIP Core module (Figure 4 (a)), eXosip is utilized to implement the SIP protocol stack. The SIP Core module supports SIP communication with other SIP UAs. This module is invoked by the Call Control module (Figure 4 (b)) to execute the call setup or teardown procedure following the standard SIP protocol. The Call Control module instructs other NCTU SIP UA modules to handle call related activities such as call establishment, call answer, call rejection, etc. The RTP Core module (Figure 4 (c)) utilizes the oRTP library [17] to implement Real-time Transport Protocol (RTP) stack under GNU Lesser General Public License (LGPL). This module builds RTP sessions between SIP UAs. The Multimedia Control module (Figure 4 (d)) supports audio functions such as wave input and output. This module plays the received voice data, and converts the user voice into RTP packets through the RTP Core module. The User Interface module (Figure 4 (e)) supports interfaces for interactions between a user and the SIP UA. The SIP Mobility module (Figure 4 (f)) supports the SIP terminal mobility. It detects the modification of local IP address by receiving an event from Winsock module, retrieves IP addresses through IP Helper API, selects new local IP address and instructs the SIP Core module to send the re-INVITE message.

Figure 4 illustrates the interaction between NCTU SIP UA modules during SIP terminal mobility. At the startup of NCTU SIP UA, the SIP Mobility module invokes the Winsock function WSAIoctl( ) with parameter "SIO_ADDRESS_LIST_CHANGE" to subscribe to the notification event for local address list modification. This list includes all IP addresses assigned to the host. Through function WSACreateEvent( ), the SIP Mobility module names

the notification event as Address-Change. Through function WSAEventSelect( ), it requests the Winsock module to trigger this event when any of the local IP addresses are modified, added, or deleted. With a polling mechanism, the SIP Mobility module detects the Address-Change event.

Suppose that an IPv6 SIP multimedia session is established between the MH and the CH. The local address list of the MH contains an IPv6 address initially. When the MH moves to another IPv6 network, the MH obtains a new IPv6 address, and the previous IPv6 address becomes no longer available. When the local address list is modified by the operating system, the Address-Change event is triggered and detected by the SIP Mobility module. Then the following steps are executed.

Steps 1 and 2. When the SIP Mobility module detects the Address-Change event, it invokes GetAdaptersAddresses( ) in IP Helper API to retrieve the local address list. If the current address bound by the SIP UA is not in the list, then another address from the list is selected as the new address of the SIP UA. Since the MH moves to another network and is assigned a new IPv6 address, the new IPv6 address is selected as the new local address.

Step 3. At the startup of the NCTU SIP UA, the eXosip library will create a UDP socket for SIP signaling [9]. The socket is bound to the local IP address. After the IP address is changed, the SIP Mobility module modifies this socket with the new IPv6 address through the function eXosip_modify_ip( ) provided by the SIP Core module.

Step 4. After the socket in eXosip is modified, the SIP Mobility module generates an

event SIPCore_IPCHANGE_NEWIP_NOTIFY to notify the Call Control module.

Steps 5 and 6. The Call Control module sends the re-INVITE request to the CH using the new IPv6 address through the SIP Core module. Then the MH receives the IPv6 200 OK response from the CH indicating that the CH accepts the re-INVITE message.

Steps 7 and 8. The MH sends an IPv6 ACK message to the CH to confirm the media session establishment. The Call Control module instructs the Multimedia Control module to suspend the RTP session and then resume the RTP session with the new IPv6 address.

After Step 8, the RTP session between the MH and the CH is re-established.

The above example shows the procedure for IPv6-only SIP terminal mobility. For IPv4-only SIP terminal mobility, the message flow is the same, except that the SIP messages are sent and received through IPv4.

2.2 RADVISION Implementation

The RADVISION SIP Toolkit is a commercial solution for SIP implementations. Compared with the eXosip library, the RADVISION SIP Toolkit provides more powerful functions, such as SIP signaling compression and the ability of delivering SIP messages over SCTP and TLS. One important feature is that the RADVISION SIP Toolkit supports IPv4 and IPv6 dual-stack, and it can bind IPv4 and IPv6 addresses concurrently for sending/receiving SIP messages. (This feature is not supported in current eXosip library.) Therefore, we utilize

the RADVISION SIP Toolkit to implemented SIP terminal mobility for heterogeneous IPv4/IPv6 networks.

Figure 5 illustrates the dual-stack SIP UA architecture. In this figure, gray boxes represent the SIP UA components we implemented based on the RADVISION SIP Toolkit. The components with white color are Winsock API and network protocols provided by the operating system.

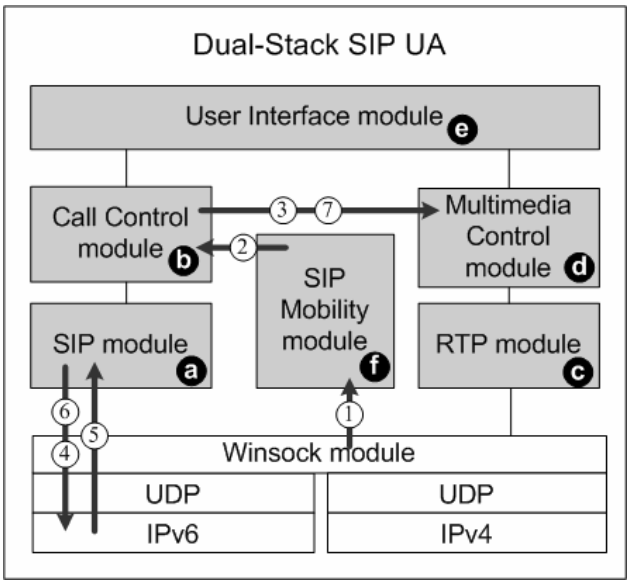


Figure 5. Dual-Stack SIP UA architecture

Compared with Figure 4, the Call Control module (Figure 5 (b)), the Multimedia Control module (Figure 5 (d)), the User Interface module (Figure 5 (e)) and the SIP Mobility module (Figure 5 (f)) provide the same functions as those described in the NCTU SIP UA. The SIP module and the RTP module also provide the same functions as the SIP Core module and the RTP Core module described in the NCTU SIP UA, but here RADVISION SIP Toolkit is utilized to implement these two modules. Compared with the NCTU SIP UA, the main differences include: (1) In the Call Control module, a flag “addrTypePrefer” indicates the preference of the IP version (IPv4 or IPv6) when both IPv4 and IPv6 connections are

available. (2) In the SIP module, the function GetAdaptersAddresses( ) in IP Helper API used to retrieve local IP addresses is replaced by the function WSAIoctl( ) with parameter “SIO_ADDRESS_LIST_QUERY” in Winsock API. Therefore the IP Helper API is not needed in this dual-stack SIP UA. (3) The function WSAIoctl( ) with parameter “SIO_ROUTING_INTERFACE_QUERY” is invoked in the SIP Mobility module to select the new local address. This function can select a proper source address from local address list to connect to the destination address. Therefore, we can use this function to automatically determine the new local address and need not manually select it.

Figure 5 illustrates the interaction between dual-stack SIP UA modules during SIP terminal mobility. At the startup of dual-stack SIP UA, the SIP Mobility module invokes the Winsock function WSAIoctl( ) with parameter "SIO_ADDRESS_LIST_CHANGE" to subscribe to the notification event for local address list modification. The details are the same as described in Section 2.1.

Suppose that an IPv4 SIP multimedia session is established between the MH and the CH. The local address list of the MH contains an IPv4 address initially. When the MH moves to an IPv6 network, the MH obtains an IPv6 address through IPv6 address autoconfiguration, and the previous IPv4 address becomes no longer available. When the local address list is modified by the operating system, the Address-Change event is triggered and received by the SIP Mobility module. Then the following steps are executed.

Step 1. When the SIP Mobility module detects the Address-Change event, it invokes WSAIoctl( ) with parameter “SIO_ADDRESS_LIST_QUERY” to retrieve the local address list. If the current address used by the SIP UA is not in the list, then WSAIoctl( ) with parameter “SIO_ROUTING_INTERFACE_QUERY” is invoked to select another

address from the list as the new address of the SIP UA. Suppose that the IPv6 address acquired from the IPv6 network is selected as the new local address.

Steps 2 and 3. The SIP Mobility module instructs the Call Control module to add the new address to send SIP messages and remove the old address. Then the Call Control module requests the Multimedia Control module to suspend the RTP session.

Steps 4 and 5. The Call Control module sends the re-INVITE request to the CH using the new IPv6 address through the SIP module. Then the MH receives the IPv6 200 OK response from the CH indicating that the CH accepts the re-INVITE message.

Steps 6 and 7. The MH sends an IPv6 ACK message to the CH to confirm the media session establishment. The Call Control module instructs the Multimedia Control module to resume the RTP session with the new IPv6 address.

After Step 7, the multimedia session is re-established through the IPv6 network.

Chapter 3
Performance Evaluation & Comparison

This chapter investigates the performance of SIP terminal mobility. As shown in Figure 6, the delays for switching a RTP session in a wireless LAN environment can be divided into the following parts [18]: D1 is the delay for radio link switching from one Access Point (AP) to another. D2 is the delay for detecting a new router and a new IP subnet after switching AP, where the MH detects that it has moved to a new subnet by detecting the IP addresses change of the host or by listening to the IPv6 Router Advertisement. D3 is the delay between when the MH activates the SIP terminal mobility procedure and when it receives the 200 OK response for the re-INVITE request. D5 is the delay between when the MH receives the 200 OK message and when the media transmission is resumed. Note that depending on the SIP implementations (to be elaborated later), the RTP session suspension is conducted in either D3 or D5. In [18], D1, D2, D3 and D4=D3+D5 are also utilized to measure the performance of SIP terminal mobility. Since both D1 and D2 are link-layer and IP-layer delays, they can be independently evaluated without affecting the application-level performance for SIP terminal mobility. This thesis will focus on D3, D4, and D5.

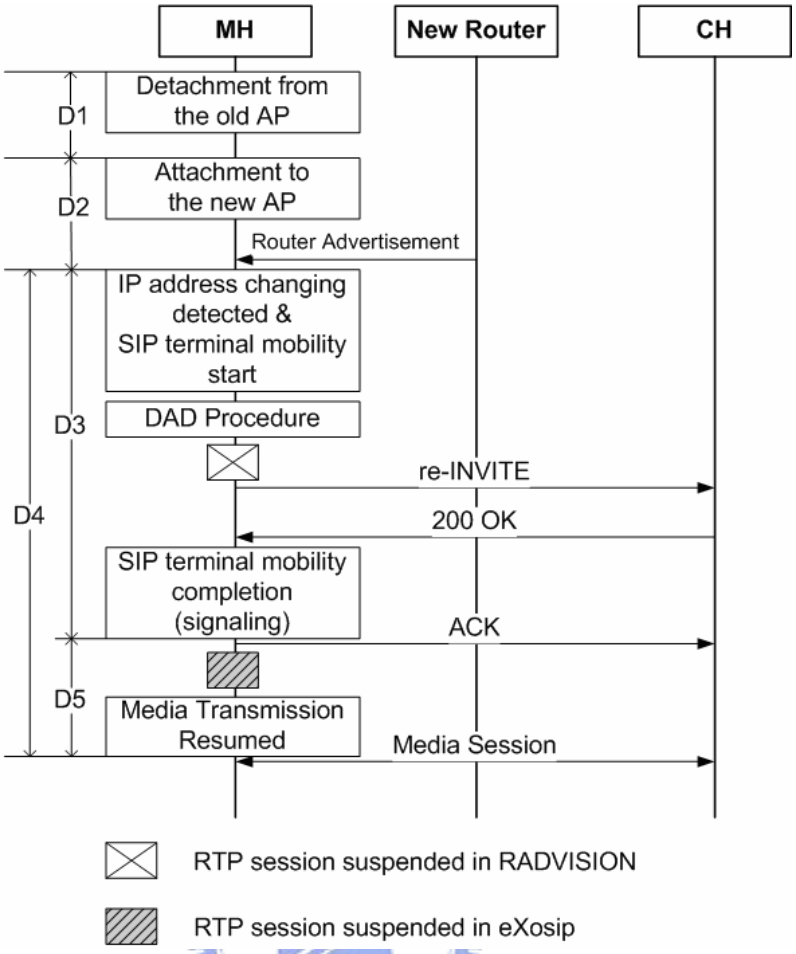


Figure 6. The delays for switching a RTP session from one AP to another AP

3.1 Performance Evaluation in IPv4 and IPv6 Networks

Figure 7 illustrates the IPv4/IPv6 experimental environment in our study. The MH and the CH are our SIP UA implementations. Two IPv4/IPv6 dual-stack routers (Router A and Router B) running FreeBSD version 6.1 are connected directly through an Ethernet cable. Router A connects to the CH through Subnet 1. Router B connects to two 802.11b APs (D-Link DWL-1000 APs) through Subnet 2 and Subnet 3, respectively. Initially, the MH in Subnet 2 establishes a SIP session with the CH in Subnet 1. Then the MH moves from Subnet 2 to

Subnet 3. SIP terminal mobility delays (i.e., D3 and D4) during the MH's movement are measured.

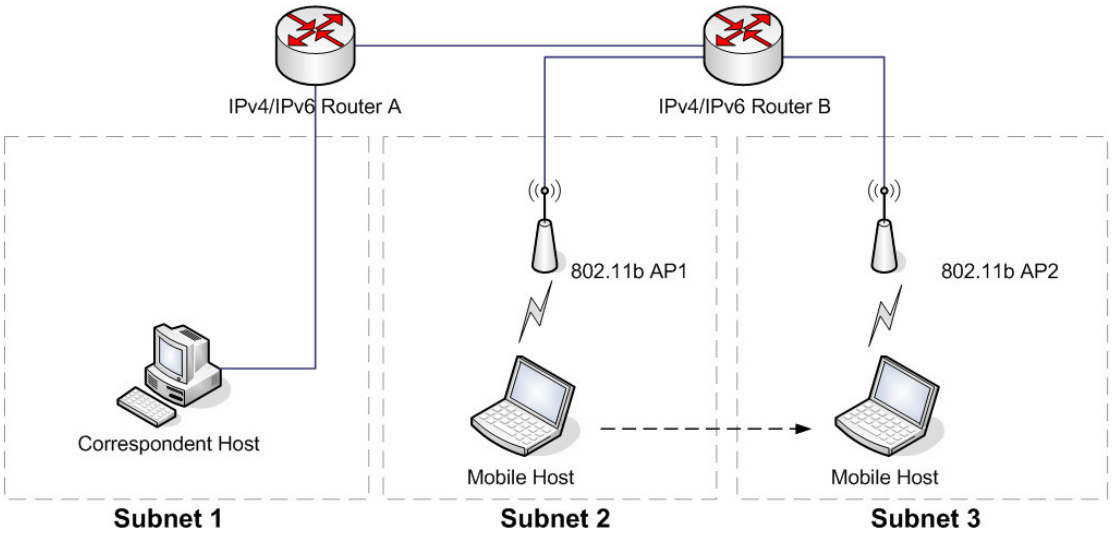


Figure 7. The experimental environment

Table 1 shows the measured D3, D4, and D5 delays for our SIP terminal mobility implementations based on RADVISION SIP Toolkit and eXosip. This table also shows the delays for SIP terminal mobility in Columbia University's SIP UA reported by N. Nakajima, A. Dutta, S. Das and H. Schulzrinne [18], which is indicated as NDDS in the table. In the table, Scenario 1 is SIP terminal mobility for IPv4-only networks, where the MH moves from one IPv4 network to another. Scenario 2 is SIP terminal mobility for IPv6-only networks, where the MH moves from one IPv6 network to another. In Scenario 2, (a) is the scenario without DAD and (b) is the scenario with DAD. Scenario 3 is SIP terminal mobility between IPv4 and IPv6 networks. In Scenario 3, (a) is SIP terminal mobility where the MH moves from an IPv4 network to an IPv6 network without DAD, (b) is the scenario for the MH moves from an IPv4 network to an IPv6 network with DAD, and (c) is SIP terminal mobility where the MH moves from an IPv6 network to an IPv4 network.

For the RADVISION implementation, we observe that Scenarios 1, 2(a), 3(a), and 3(c) have almost the same D4 performance. In Scenario 2(b) and 3(b), the D3 delays are about 1.25 seconds longer than those measured in Scenario 2 (a) and 3(a). This extra delay is contributed by DAD. Therefore if the DAD procedure is not executed, the delays for IPv4/IPv6 scenarios are close to those for the IPv4-only and IPv6-only cases. Table 2 shows the standard derivations for D3 and D4 in the RADVISION implementation. The standard derivations for all scenarios except for those with DAD are roughly the same. From the above discussion, we conclude that SIP terminal mobility implementation between IPv4 and IPv6 networks can be as efficient as those for IPv4-only and IPv6-only networks.

Table 1. D3 and D4 delays for RADVISION, eXosip, and NDDS implementations

Scenario	Implementation	D3 (ms)	D5 (ms)	D4 (ms)
Scenario 1 : IPv4-only	RADVISION	102.5	11.2	113.7
	eXosip	58.1	173.6	231.7
Scenario 2 (a) : IPv6-only (without DAD)	RADVISION	102.4	10.7	113.1
	eXosip	55.2	168.2	223.4
	NDDS	161.6	257.0	418.6
Scenario 2 (b) : IPv6-only (with DAD)	RADVISION	1346.6	10.6	1357.2
	eXosip	1310.6	168.4	1479.0
	NDDS	3932.2	255.5	4187.7
Scenario 3 (a) : IPv4 to IPv6 (without DAD)	RADVISION	102.0	10.2	112.2
Scenario 3 (b) : IPv4 to IPv6 (with DAD)	RADVISION	1345.0	10.6	1356.4
Scenario 3 (c) : IPv6 to IPv4	RADVISION	102.1	11.1	113.2

Table 2. D3 and D4 standard derivations for the RADVISION implementation

Scenario	Standard derivation for D3 (ms)	Standard derivation for D4 (ms)
Scenario 1 : IPv4-only	19.4	20.2
Scenario 2 (a) : IPv6-only (without DAD)	18.0	19.6
Scenario 2 (b) : IPv6-only (with DAD)	147.5	147.5
Scenario 3 (a) : IPv4 to IPv6 (without DAD)	20.6	21.4
Scenario 3 (b) : IPv4 to IPv6 (with DAD)	146.9	146.9
Scenario 3 (c) : IPv6 to IPv4	19.5	20.4

Table 1 also shows that the D3 delays for the eXosip implementation are shorter than those for the RADVISION implementation. On the contrary, the D4 delays of the eXosip implementation are longer. This is due to the different ways in resetting the RTP connection. The SIP UA based on RADVISION resets the RTP connection before sending the SIP re-INVITE message (see ⊠ in Figure 6) so its overhead is included in D3. On the other hand, the SIP UA based on eXosip resets RTP connection after sending the re-INVITE message (see ▨ in Figure 6) so its overhead is included in D5. In other words, if we consider the overall delay D4, the RADVISION implementation is better than the eXosip implementation.

Now let us take a look at the NDDS implementation. N. Nakajima, A. Dutta, S. Das and H. Schulzrinne (NDDS) evaluated the performance of SIP terminal mobility in IPv6-only networks. The configuration of the NDDS experimental environment is the same as that illustrated in Figure 7 except that the routers are IPv6-only routers. The MH and the CH are Columbia University's SIP UAs running on Linux [18]. Table 1 shows the results of NDDS experiments. In NDDS Scenario 2 (a), D3 is 161.6ms and D5 is 257.0ms. These delays are higher than those in both our RADVISION and eXosip implementations. These long delays may be due to the fact that the Columbia University's SIP UA was implemented by Tcl/Tk

that provides a friendly programming environment with higher object code execution overhead [19].

In NDDS Scenario 2(b), D3 is 3932.2ms and D5 is 255.5ms. In this scenario, there is an extra delay from Neighbor Unreachability Detection (NUD) in Linux [20]. With NUD mechanism, when the MH moves to a new subnet, the MH still tries to send the packets through the old access router until the unreachability of the old router is detected. The delay from the NUD contributes 3770.6 ms to D3. In our experiments (for both RADVISION and eXosip implementations), the SIP UAs are running on Windows XP instead of Linux, so NUD is not executed. Therefore, our experiments show shorter delay compared with NDDS implementation on Linux.

The D3, D4, and D5 delays in NDDS scenarios show the similar variation as those in our eXosip experiments. Therefore it is reasonable to conclude that the results of both studies are consistent for the IPv6-only scenarios.

3.2 Interoperability

In this section we shall show the experimental results between the NCTU SIP UA and other SIP UAs. The test is not performed on the RADVISION implementation since most SIP UAs do not support IPv4/IPv6 dual-stack. In this experiment, we use the eXosip-based NCTU SIP UA as the MH, and select one SIP UA (softphone or hardphone) to be the CH. The MH and CH are connected to the same 100Mbps switching hub. Initially, the MH establishes a SIP session with the CH in the same subnet. Then the MH changes its IP address and SIP terminal mobility is executed. Table 3 shows those SIP UAs and their experimental results. Among

those UAs, NCTU SIP UA, Windows Messenger, and X-Lite UA are sofphones. Snom200, Cisco 7940, InnoMedia video phone, and Pingtel are hardphones. Notice that even though the MH is always NCTU SIP UA, the delay of D3 is quite different for different CH. The reason is that after receiving a re-INVITE request, each SIP UA requires different time to handle the request and then generates the SIP 200 OK response.

From Table 3, the results of NCTU SIP UA and Windows Messenger 5.1 are very close, and the X-Lite UA is a little longer (about 11% for D4). In the results of hardphones, the delay is obviously longer than those sofphones. We consulted the engineers in InnoMedia Corporation, and they believe that it is caused by the time spent on SDP parsing in the protocol stack. Because InnoMedia video phone supports both video and audio transmission during SIP conversation, the SDP contents contain video media description. Therefore this extra complexity increases the delay slightly. We also perform the same experiment on Cisco 7940 SIP hardphones with different firmware versions. In firmware version 7.5, the delay of sending 200 OK response is longer than that in firmware version 5.3. The reason is that the newer version applies more rigorous rules in checking the SDP contents. For example, in firmware version 7.5 when the re-INVITE is received, the SDP version in session identifier field (the third field in the “o=” line) must be verified to see whether it is incremented by 1. Certainly this increases the delay, too.

Table 3. D3 and D4 delays between NCTU SIP UA and other SIP UAs

Devices Under Test		D3 (ms)	D4 (ms)	Media resumption delay (D4) compared to NCTU SIP UA
SIP UA	Version			
NCTU SIP UA (IPv4)	1.1	38.2	214.4	100.00%
Windows Messenger	5.1.0680	38.2	214.3	99.95%
X-Lite UA	2.0 build 1103	50.2	238.4	111.19%
Snom 200 hardphone	1.16x 4904	94.8	270.9	126.35%
Cisco 7940 hardphone	5.3	151.3	340.2	158.68%
Cisco 7940 hardphone	7.5	230.2	404.4	188.62%
InnoMedia video phone	2.4.17	173.1	356.1	166.09%
Pingtel hardphone	2.1.11.24	195.0	370.6	172.85%

Chapter 4
Conclusions

This thesis investigates SIP terminal mobility and its implementations. In previous studies, SIP terminal mobility was discussed and implemented for homogeneous IPv4-only or IPv6-only networks. In this thesis, we introduce SIP terminal mobility for heterogeneous IPv4/IPv6 networks and its implementation. The designs of protocol architecture and the implementations based on two popular APIs, eXosip and RADVISION, are described. Their performance is also measured from empirical experiment. In IPv6-only network with DAD, it takes 1479.0ms for the eXosip implementation and 1357.2ms for the RADVISION implementation during the SIP terminal mobility procedure, so this may be short enough to support daily conversations. Furthermore if the DAD procedure is not executed, the delay is reduced to 223.4ms and 113.1ms respectively. It is obvious that the delay from the DAD procedure becomes the bottleneck for SIP terminal mobility in IPv6 networks.

In the eXosip implementation, the interoperability testing of SIP terminal mobility among SIP UAs is demonstrated. The results show that the delay of SIP terminal mobility does not only depend on the MH, but also on the CH. After receiving a re-INVITE request, each SIP UA needs a period of time to run some internal processing (e.g. SDP parsing) before it replies a 200 OK response, so the delay of SIP terminal mobility differs divergently. The information provided by the manufacturer shows that one of the major factors is the complexity required in processing the SIP header fields and SDP contents.

The performance of SIP terminal mobility between IPv4 and IPv6 networks is also measured in the RADVISION implementation. The delays for IPv4/IPv6 scenarios are close to those for the IPv4-only and IPv6-only cases. These results show that the performance of IPv4/IPv6 SIP terminal mobility can be as efficient as those for IPv4-only and IPv6-only networks. From the above observations, it shows that SIP terminal mobility can be efficiently supported not only in IPv4-only and IPv6-only networks, but also between IPv4 and IPv6 networks.

Bibliography

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler , “SIP: Session Initiation Protocol”, IETF RFC 3261, June 2002.

[2] H. Schulzrinne and E. Wedlund, “Application-layer mobility using SIP”, ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 4, Number 3, pp.47-57, July 2000.

[3] C.-H. Yeh, Q. Wu, Y.-B. Lin, "SIP Terminal Mobility for both IPv4 and IPv6", The 8th International Workshop on Multimedia Network Systems and Applications (MNSA) in IEEE 26th International Conference on Distributed Computing Systems (ICDCS 2006), Lisbon, Portugal, July 4-7, 2006.

[4] S. Thomson and T. Narten, “IPv6 stateless address autoconfiguration”, IETF RFC 2462, December 1998.

[5] J.-H. Weng, “Design and Implementation of SIPv6 Translator”, master’s thesis, National Chiao Tung University, Department of Computer Science and Information Engineering, July 2005

[6] G. Tsirtsis and P. Srisuresh, “Network Address Translation - Protocol Translation (NAT-PT)”, IETF RFC 2766, February 2000.

[7] W.-E. Chen, Q. Wu, and Y.-B. Lin, “Design of SIP Application Level Gateway for IPv6 Translation”, Journal of Internet Technology (JIT) Special Issue on IPv6. Vol. 5 No. 2, 2004.

[8] IEEE, "Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority", http://grouper.ieee.org/groups/msc/MSC200407/OnlineTutorials/EUI64.htm, March 1997.

[9] The eXtended osip library, http://savannah.nongnu.org/ projects/exosip/

[10] The sipX libraries, http://www.sipfoundry.org/sipX/index.html

[11] The RADVISION SIP Toolkit, http://www.radvision.com/

[12] The Signalware SIP library, http://www.ulticom.com/html/products/signalware-sip.asp

[13] The jSIP library, http://jsip.sourceforge.net/

[14] The JAIN-SIP library, https://jain-sip.dev.java.net/

[15] GNU oSIP library, http://www.gnu.org/software/osip/

[16] L.-Y. Wu, M.-H. Tsai, Y.-B. Lin, and R.-S. Yang, "A Client-Side Design and Implementation for Push to Talk over Cellular Service". Accepted and to appear in Wireless Communications and Mobile Computing Journal.

[17] oRTP – a Real-time Transport Protocol stack under LGPL, http://linphone.org/ortp/

[18] N. Nakajima, A. Dutta, S. Das, and H. Schulzrinne, "Handoff Delay Analysis and Measurement for SIP based Mobility in IPv6", IETF International Conference on Computers and Communications, Anchorage Alaska, USA, May 2003.

[19] Tcl/Tk, http://www.tcl.tk/

[20] T. Narten, E. Nordmark, and W. Simpson, "Neighbor Discovery for IP version 6 (IPv6)", IETF RFC 2461, December 1998.

Appendix A
The SIP Mobility Module Program in the eXosip Implementation

Appendix A lists the source code of the SIP Mobility module program in the eXosip implementation. The SIP Mobility module is implemented as a C++ program, which consists of the header files **`AddressChange.h`** (Appendix A.1), **`CSIPUACore.h`** (Appendix A.3), and the program files **`AddressChange.cpp`** (Appendix A.2), **`CSIPUACore.cpp`** (Appendix A.4). In **`CSIPUACore.cpp`**, only the portion for SIP Mobility module is listed. In **`AddressChange.cpp`,** a class CAddressChange is implemented which utilizes the function WSAIoctl( ) in Winsock API to detec the Address-Change event.

A.1 `AddressChange.h`

```
01      #if !defined(AFX_ADDRESSCHANGE_H__088D2025_6860_4E36_A3BA_B39C8F03295F__INCLUDED_)
02      #define AFX_ADDRESSCHANGE_H__088D2025_6860_4E36_A3BA_B39C8F03295F__INCLUDED_
03
04      #if _MSC_VER > 1000
05
06      #include <winsock2.h>
07      #include <windows.h>
08      //#include <iphlpapi.h>
09      #include <tchar.h>
10
11      #include "CriticalSection.h"
12      using ADDRESSCHANGE::CAddressChangeCriticalSection;
13
14      #include <vector>
15
16      using namespace std;
17      #pragma comment(lib, "ws2_32.lib")
18      #pragma   comment(lib, "iphlpapi.lib")
19      #endif // _MSC_VER > 1000
20
```

```
21      namespace       ADDRESSCHANGE
22      {
23
24
25      typedef vector<TCHAR*> STRINGVECTOR,*PSTRINGVECTOR;
26      typedef vector<TCHAR*>::iterator STRINGVECTORIT,*PSTRINGVECTORIT;
27      typedef vector<DWORD> DWORDVECTOR,*PDWORDVECTOR;
28      typedef vector<DWORD>::iterator DWORDVECTORIT,*PDWORDVECTORIT;
29
30      typedef     void(*NotifyFunction)(void *pri);
31      typedef struct
32      {
33          NotifyFunction func;
34          void *pri;
35      }CallbackEntry, *PCallbackEntry;
36
37
38      class CAddressChange
39      {
40      private:
41          static CAddressChangeCriticalSection m_cs;
42          static HANDLE m_ipthreadv6;
43          static vector<CallbackEntry> m_IpChangeNotifyVector;
44          static int m_ipthreadstopflag;
45          static ULONG __stdcall Ipv6ChangeNotifyThread(void* pri);
46      public:
47          CAddressChange();
48
49          static int StartIpChangeNotify();
50          static void StopIpChangeNotify();
51
52          static void AddIpChangeNotify(CallbackEntry cbEntry);          //Not multi-thread safe
53          static void RemoveIpChangeNotify(CallbackEntry cbEntry);       //Not multi-thread safe
54          virtual ~CAddressChange();
55      };
56
57      }
58
59      #endif // !defined(AFX_ADDRESSCHANGE_H__088D2025_6860_4E36_A3BA_B39C8F03295F__INCLUDED_)
```

A.2 AddressChange.cpp

```
001     #include "AddressChange.h"
002
003     namespace ADDRESSCHANGE
004     {
005
006     //////////////////////////////////////////////////////////////////////
007     // Static members implementation
008     //////////////////////////////////////////////////////////////////////
009     CAddressChangeCriticalSection CAddressChange::m_cs;
010     HANDLE CAddressChange::m_ipthreadv6 = NULL;
011     vector<CallbackEntry> CAddressChange::m_IpChangeNotifyVector;
012     int CAddressChange::m_ipthreadstopflag = 0;
013
014     //////////////////////////////////////////////////////////////////////
015     // Construction/Destruction
016     //////////////////////////////////////////////////////////////////////
017
018     CAddressChange::CAddressChange()
019     {
020
021     }
022
023     CAddressChange::~CAddressChange()
024     {
025
```

```
026         }
027         ////////////////////////////////////////////////////////////////////
028         // Static methods implementation
029         ////////////////////////////////////////////////////////////////////
030
031         int CAddressChange::StartIpChangeNotify()
032         {
033             DWORD dwid;
034
035             m_cs.Lock();
036             if(NULL == m_ipthreadv6)
037             {
038
039                     m_ipthreadv6 =
CreateThread(NULL,NULL,Ipv6ChangeNotifyThread,NULL,CREATE_SUSPENDED,&dwid);
040
041                     if(NULL == m_ipthreadv6)
042                     {
043                             return 0;
044                     }
045
046                     m_ipthreadstopflag = 0;
047                     ResumeThread(m_ipthreadv6);
048             }
049
050             m_cs.Unlock();
051
052             return 1;
053         }
054
055         void CAddressChange::StopIpChangeNotify()
056         {
057             m_cs.Lock();
058             if(NULL != m_ipthreadv6)
059             {
060                     m_ipthreadstopflag = 1;
061                     WaitForSingleObject(m_ipthreadv6,INFINITE);
062                     DeleteObject(m_ipthreadv6);
063                     m_ipthreadv6 = NULL;
064             }
065             m_cs.Unlock();
066         }
067         void CAddressChange::AddIpChangeNotify(CallbackEntry cbEntry)
068         {
069             m_cs.Lock();
070             m_IpChangeNotifyVector.push_back(cbEntry);
071             m_cs.Unlock();
072         }
073         void CAddressChange::RemoveIpChangeNotify(CallbackEntry cbEntry)
074         {
075             m_cs.Lock();
076             for(vector<CallbackEntry>::iterator it = m_IpChangeNotifyVector.begin();
077                     it != m_IpChangeNotifyVector.end() ;
078                     it++)
079             {
080                     if(it->func == cbEntry.func && it->pri == cbEntry.pri)
081                     {
082                             m_IpChangeNotifyVector.erase(it);
083                             break;
084                     }
085             }
086             m_cs.Unlock();
087         }
088
089         ULONG CAddressChange::Ipv6ChangeNotifyThread(void* pri)
090         {
091
092             int                     err;
093             SOCKET          socketHandle;
094             unsigned long           param;
095             int                             inBuffer;
096             int                             outBuffer;
097             DWORD                           outSize;
098             WSAEVENT NewEvent;
099             WSADATA         wsaData;
100
```

```
101
102             err = WSAStartup( MAKEWORD( 2,2 ), &wsaData );
103
104             while(!m_ipthreadstopflag)
105             {//1ST loop bracket
106
107                     socketHandle = socket( AF_INET6, SOCK_DGRAM, IPPROTO_UDP );
108
109                     param = 1;
110                     err = ioctlsocket( socketHandle, FIONBIO, &param );
111
112
113                     inBuffer        = 0;
114                     outBuffer       = 0;
115                     err = WSAIoctl( socketHandle, SIO_ADDRESS_LIST_CHANGE, &inBuffer, 0, &outBuffer, 0, &outSize,
NULL, NULL );
116
117                     NewEvent = WSACreateEvent();
118                     err = WSAEventSelect( socketHandle, NewEvent, FD_ADDRESS_LIST_CHANGE );
119
120                     while(!m_ipthreadstopflag)
121                     {
122
123                             if ( WaitForSingleObject(NewEvent, 100) == WAIT_OBJECT_0 )
124                             {
125                                     for(vector<CallbackEntry>::iterator it = m_IpChangeNotifyVector.begin(); it !=
m_IpChangeNotifyVector.end();it++)
126                                     {
127                                             it->func(it->pri);
128                                     }
129                                     ResetEvent(NewEvent);
130                                     break;
131                             }
132                             else
133                             {//time out
134
135                             }
136                     }
137             }
138             return 0;
139         }
140
141
142         }//End namespace
```

A.3 CSIPUACore.h

```
001         #ifndef __CSIPUACore_H__
002         #define __CSIPUACore_H__
003
004         #include<interCommunicationClass.h>
005         #include<eXosip/eXosip.h>
006         #include<eXosip/eXosip_cfg.h>
007
008         #include<windows.h>
009         #include<time.h>
010
011         typedef struct IPAddresses{
012             char *IPAddr;
013             int INETType;           //0 for IPv6; 1 for IPv4
014             int scope;      //0 for Global;
015                                     //1 for Global -6to4 (IPv6 only);
016                                     //2 for Site-Local and IPv4 private addr.;
017                                     //3 for Link-Local
018             int preferOrder;        //set by IPSelection() or IPSelectionWithDestination() for sorting convenience
019             int priority;     //1 for tunnel interface;
020                                         //2 for usual
```

```
021         struct IPAddresses *next;
022     }IPAddresses;
023
024     typedef enum natTraversalType{
025         AUTO_DETECT_NAT,
026         NO_NAT,
027         USE_UPNP,
028         USE_STATIC_ASSIGN
029     }natTraversalType;
030
031     typedef enum internetFamily{
032         AUTO_DETECT_INET,
033         USE_IPV6,
034         USE_IPV4
035     }internetFamily;
036
037     typedef unsigned char flag;
038
039     typedef enum SIPCoreEvent{
040         SIPCore_THREAD_CREATE_FAILED,   //0
041
042         SIPCore_EXOSIP_INITIATE_FAILED,    //1
043
044         SIPCore_REGISTRATION_NEW_OK,    0//2
045         SIPCore_REGISTRATION_NEW_FAILED_NEED_AUTHENTICATION,     //3
046         SIPCore_REGISTRATION_NEW_FAILED_CHECK_CONFIGURATION,    //4
047         SIPCore_REGISTRATION_REFRESH_OK,     //5
048         SIPCore_REGISTRATION_REFRESH_FAILED,         //6
049         SIPCore_UNREGISTRATION_OK,         //7
050         SIPCore_UNREGISTRATION_FAILED,  //8
051
052         SIPCore_SEND_INVITE_FAILED_INIT,  //9
053         SIPCore_SEND_INVITE_FAILED_UNKOWN,   //10
054         SIPCore_SEND_INVITE_FAILED_CALLEE_BUSY,     //11
055         SIPCore_SEND_INVITE_FAILED_CALLEE_NOTFOUND,    //12
056         SIPCore_SEND_INVITE_FAILED_TIMEOUT,   //13
057         SIPCore_SEND_INVITE_FAILED_NO_CODEC_SUPPORT, //14
058         SIPCore_SEND_INVITE_FAILED_CANNOT_SEND_ACK,    //15
059         SIPCore_SEND_INVITE_RINGING,       5//16
060         SIPCore_SEND_INVITE_200OK,         6//17
061
062
063         SIPCore_RECV_INVITE_NEW,    //18
064         SIPCore_RECV_INVITE_ACK,    //19
065         SIPCore_RECV_INVITE_CANCEL,        //20
066         SIPCore_RECV_INVITE_TIMEOUT,       2//21
067         SIPCore_RECV_INVITE_SEND_180_FAILED, //22
068         SIPCore_RECV_INVITE_SEND_200_FAILED, //23
069         SIPCore_RECV_INVITE_FAILED_NO_CODEC_SUPPORT, //24
070
071         SIPCore_TERMINATE_CALL_OK,         //25
072         SIPCore_TERMINATE_CALL_FAILED,  //26
073         SIPCore_CALL_TERMINATED,    //27
074
075         SIPCore_SEND_INFO_OK,         //28
076         SIPCore_SEND_INFO_FAILED_INIT,     //29
077         SIPCore_SEND_INFO_FAILED_UNKNOWN,   //30
078         SIPCore_RECV_INFO_NEW,      //31
079
080         SIPCore_SEND_MESG_OK,       //32
081         SIPCore_SEND_MESG_FAILED_INIT,   //33
082         SIPCore_SEND_MESG_FAILED_TIMEOUT,    //34
083         SIPCore_SEND_MESG_FAILED_UNKNOWN, //35
084         SIPCore_RECV_MESG_NEW,    //36
085
086         //=========================================================
087         //add by chyei
088         SIPCore_RECV_REINVITE,         //37
089         SIPCore_IPCHANGE_NOIP_NOTIFY,
090         SIPCore_IPCHANGE_NEWIP_NOTIFY,
091         SIPCore_IPCHANGE_ORGIP_NOTIFY  //?
092         //=========================================================
093         //...
094     }SIPCoreEvent;
095
096     unsigned __stdcall backgroundProcess( void *argument );
```

```
097         /* a thread that runs in background,
098           * it's responsible for receiving events and do registration refresh
099           */
100         unsigned __stdcall tryToReceiveForTesting( void *argument );
101         /* a temporary thread that is for tesing if there's SIP service on a remote ip address
102           */
103
104         class CSIPUACore :public CInterCommunicationClass{
105             private:
106         #ifdef BEAN_DEBUG
107                     void ( __cdecl *m_cb_event )( const SIPCoreEvent );
108         #else
109         //          HWND m_hWnd;    //lywu's window (UI) handle
110         //  replaced by callControlPointer;
111         #endif
112                     CRITICAL_SECTION cs__callControlPointer;
113                     CRITICAL_SECTION cs__floorControlPointer;
114                     CRITICAL_SECTION cs__GLMSControlPointer;
115
116                     CInterCommunicationClass *_callControlPointer;      //lywu's call control pointer
117                     CInterCommunicationClass *_floorControlPointer;     //tsaimh's floor control pointer
118                     CInterCommunicationClass *_GLMSControlPointer;  //stephon's GLMS control pointer
119                     char *_GLMSSipuri;//stephon's GLMS sipuri
120
121                     natTraversalType _natTraversalType;
122                     internetFamily _internetFamily;
123                     flag forceINETFlag;
124                     flag forceNATFlag;
125                     char *sipuri;  //sip:xxx@yyy.zzz
126                     char *outboundProxy;        //yyy.zzz
127                     int sipPort;
128                     //information for one call
129                     flag isUACFlag;       //if is a caller in the existing call
130                     char *rtpLocalIP, *rtpRemoteIP;
131                     int rtpLocalAudioPort, rtpNatExternalAudioPort, rtpRemoteAudioPort;
132                     int rtpLocalVedioPort, rtpNatExternalVedioPort, rtpRemoteVedioPort;
133                     int rtpRemotePayloadType;         //number of payload type is referencing to RFC-3551, '-1' for no support or
init
134
135                     int callID, dialogID;
136
137                     struct registerInfo{
138                             int registerID;
139                             int expireTime;
140                             time_t registerTime;
141                             char *ToHeader;      //<sip:xxx@yyy.zzz>
142                             char *registra;      //sip:yyy.zzz
143                             int numberOfRegisterTimes;        //how many times this registerID have sent REGISTER
144                     }registerInformation;
145
146                     char *hostIPAddress, *natExternalIPAddress;
147                     int hostSIPPort, natExternalSIPPort;
148         //          int hostRTPPort, natExternalRTPPort;
149                     //what follows are for static assignment of NAT Traversal
150                     int hostRTPPortBegin, natExternalRTPPortBegin;
151                     int hostRTPPortEnd, natExternalRTPPortEnd;
152
153                     HANDLE threadHandle;
154                     char *_subject;
155                     char *_callerURI;
156
157                     internetFamily detectINET( void );
158                     natTraversalType detectNAT( void );
159                     int getRTPLocalAudioPort( void );
160                     int getRTPLocalVedioPort( void );
161                     IPAddresses *getAllHostIPAddresses( const int INET, const int scope );
162                     IPAddresses *getAllRemoteHostIPAddresses( const char *IP, const char *port );
163                     void aCallTerminatedVariablesReset( void );
164                     int doUPnPGetAPort( const int internalPort, int *externalPort );
165                     int doUPnPDeleteAPort( const int externalPort );
166                     int doIPSelection( char *IPAddress, const int size );
167                     int doIPSelectionWithDestination( const char *destination, char *sourceIPAddress, const int sourceSize, char
*destinationIPAddress, const int destinationSize );
168                     int notAnIPv4Address( const char *IPAddress );
169                     int notPrivateIPv4Address( const char *IPAddress );
170                     int unUseableHostPort( const int port );
```

```
171                     int unUseableExternalPort( const int port );
172                     void freeIPAddresses( IPAddresses *pointer );
173                     int decideTheScope( const struct sockaddr *addr );
174             public:
175                     //==========
176                     //added by chyei
177                     //
178                     static void __declspec(dllexport) performanceLog(const char *string);
179
180                     HANDLE tryToGetIPAddressHandle;
181                     unsigned int ipthreadID;
182                     bool ifANYAddress;
183                     bool ifhostIPAddress;
184
185                     int __declspec(dllexport) doREINVITE();
186                     int __declspec(dllexport) modifyeXosipIP(int, int, int);
187
188                     static unsigned __stdcall tryToGetIPAddress( void *argument );
189
190                     void IpChangeHandler(int Param);
191                     void dealRecvReInvite( const eXosip_event_t *eXosipEvent );
192                     void AppSIPMobilityGetAdaptersAddresses();
193
194                     //==========
195
196                     inline int getRegisterExpiredTime( time_t *expiredTime );
197                     void dealRegisterSuccess( const eXosip_event_t *eXosipEvent );
198                     void dealRegisterFailure( const eXosip_event_t *eXosipEvent );
199                     void dealSendInvite_100( const eXosip_event_t *eXosipEvent );
200                     void dealSendInvite_180( const eXosip_event_t *eXosipEvent );
201                     void dealSendInvite_200( const eXosip_event_t *eXosipEvent );
202                     void dealSendInvite_404( const eXosip_event_t *eXosipEvent );
203                     void dealSendInvite_408( const eXosip_event_t *eXosipEvent );
204                     void dealSendInvite_486( const eXosip_event_t *eXosipEvent );
205                     void dealSendInivte_4xx5xx6xx( const eXosip_event_t *eXosipEvent );
206                     void dealRecvInvite_new( const eXosip_event_t *eXosipEvent );
207                     void dealRecvInvite_ack( const eXosip_event_t *eXosipEvent );
208                     void dealRecvInvite_cancel( const eXosip_event_t *eXosipEvent );
209                     void dealRecvInvite_timeout( const eXosip_event_t *eXosipEvent );
210                     void dealRecvBye( const eXosip_event_t *eXosipEvent );
211                     void dealSendInfo_200( const eXosip_event_t *eXosipEvent );
212                     void dealSendInfo_timeout( const eXosip_event_t *eXosipEvent );
213                     void dealSendInfo_4xx5xx6xx( const eXosip_event_t *eXosipEvent );
214                     void dealRecvInfo( const eXosip_event_t *eXosipEvent );
215                     void dealSendMesg_200( const eXosip_event_t *eXosipEvent );
216                     void dealSendMesg_timeout( const eXosip_event_t *eXosipEvent );
217                     void dealSendMesg_4xx5xx6xx( const eXosip_event_t *eXosipEvent );
218                     void dealRecvMesg( const eXosip_event_t *eXosipEvent );
219                     void echoEvent( const SIPCoreEvent event );
220             #ifdef BEAN_DEBUG
221                     __declspec(dllexport) CSIPUACore( void ( __cdecl *cb_event )( const SIPCoreEvent ) );
222             #else
223                     __declspec(dllexport) CSIPUACore();
224             #endif
225                     __declspec(dllexport) ~CSIPUACore();
226                     int __declspec(dllexport) initiation( void );
227                     int __declspec(dllexport) termination( void );
228                     int __declspec(dllexport) setNetworkConfiguration( const int setInternetFamily, const int
setNatTraversalType, const char **configurations );
229                             int __declspec(dllexport) setSIPURI( const char *username, const char *hostname );
230                             int __declspec(dllexport) setOutboundProxy( const char *proxyIPorDomainName );
231                             void __declspec(dllexport) setCallControlPointer( CInterCommunicationClass
*callControlPointer );
232                             void __declspec(dllexport) setFloorControlPointer( CInterCommunicationClass
*floorControlPointer );
233                             void __declspec(dllexport) setGLMSControlPointer( CInterCommunicationClass
*GLMSControlPointer, char *GLMSSipuri );
234                             int __declspec(dllexport) getRTPLocalRemoteIPPort( int *INETFamily, int *localAudioPort, int
*localVedioPort, char *localIP, const int localIPSize, int *remoteAudioPort, int *remoteVedioPort, char *remoteIP, const int
remoteIPSize );
235                             char __declspec(dllexport) *getCallSubject( void );
236                             char __declspec(dllexport) *getCallerURI( void );
237                             int __declspec(dllexport) getRTPRemotePayloadType( void );
238                             void __declspec(dllexport) doREGISTER( const char *registra, const int expireTime );
239                             void __declspec(dllexport) doUNREGISTER( const char *registra );
240                             void __declspec(dllexport) doINVITE( int phoneType, const char *calleeSIPURI, const char
```

```
*subject, int noSDPFlag );
241                                 int __declspec(dllexport) doBYE( void );
242                                 int __declspec(dllexport) acceptCall( void );
243                                 int _declspec(dllexport) rejectCall( void );
244                                 virtual int _declspec(dllexport) doINFO( const char *contentType, const char
*contentOfINFOBody );
245                                 virtual int _declspec(dllexport) doMESSAGE( const char *toURI, const char
*contentOfMESSAGEBody );
246                     };
247
248                     #endif
```

A.4 CSIPUACore.cpp (partial code)

```
001     #define _WSPAPI_COUNTOF
002
003     //#define IPV4_TEST
004     #define IPV6_TEST
005
006     #include<winsock2.h>
007     #include<ws2tcpip.h>
008     #include<Iphlpapi.h>
009     #include<Iptypes.h>
010     #include<windows.h>
011     #include<natupnp.h>
012     #include<process.h>
013     #include<string.h>
014     #include<stdlib.h>
015     #include"AddressChange.h"
016     using namespace ADDRESSCHANGE
017
018     TCallback<CSIPUACore> IPChangedCallback;
019     bool noAddress = false;
020
021
022     CSIPUACore::CSIPUACore()
023     #endif
024     {
025         unsigned int threadID;
026
027     #ifdef BEAN_DEBUG
028         m_cb_event = cb_event;
029     #else
030     //   m_hWnd = hWnd;
031     //   replaced by callControlPointer
032     #endif
033         //critical section
034         InitializeCriticalSection( &cs__callControlPointer );
035         InitializeCriticalSection( &cs__floorControlPointer );
036         InitializeCriticalSection( &cs__GLMSControlPointer );
037         InitializeCriticalSection( &cs_CSIPUACore_SIPCoreStop );
038         InitializeCriticalSection( &cs_CSIPUACore_eXosipOn );
039
040         EnterCriticalSection( &cs__floorControlPointer );
041         _floorControlPointer = NULL;
042         LeaveCriticalSection( &cs__floorControlPointer );
043         EnterCriticalSection( &cs__GLMSControlPointer );
044         _GLMSControlPointer = NULL;
045         _GLMSSipuri = NULL;
046         LeaveCriticalSection( &cs__GLMSControlPointer );
047
048         _natTraversalType = AUTO_DETECT_NAT;
049         _internetFamily = AUTO_DETECT_INET;
050         forceINETFlag = 0;
051         forceNATFlag = 0;
052         sipuri = NULL;
053         outboundProxy = NULL;
054         sipPort = 0;
```

```
055             isUACFlag = 0;
056             rtpLocalIP = rtpRemoteIP = NULL;
057             rtpLocalAudioPort = rtpRemoteAudioPort = 0;
058             rtpLocalVedioPort = rtpRemoteVedioPort = 0;
059             rtpNatExternalAudioPort = rtpNatExternalVedioPort = 0;
060             rtpRemotePayloadType = -1;
061             callID = dialogID = 0;
062             registerInformation.expireTime = 0;
063             registerInformation.registerID = -1;
064             registerInformation.registerTime = 0;
065             registerInformation.registra = NULL;
066             registerInformation.ToHeader = NULL;
067             registerInformation.numberOfRegisterTimes = 0;
068
069             hostIPAddress = natExternalIPAddress = NULL;
070             hostSIPPort = natExternalSIPPort = 0;
071         //  hostRTPPort = natExternalRTPPort = 0;
072             hostRTPPortBegin = natExternalRTPPortBegin = 0;
073             hostRTPPortEnd = natExternalRTPPortEnd = 0;
074
075             _subject = NULL;
076             _callerURI = NULL;
077
078             CSIPUACore_UACore = this;
079             EnterCriticalSection( &cs_CSIPUACore_eXosipOn );
080             CSIPUACore_eXosipOn = 0;
081             LeaveCriticalSection( &cs_CSIPUACore_eXosipOn );
082             EnterCriticalSection( &cs_CSIPUACore_SIPCoreStop );
083             CSIPUACore_SIPCoreStop = 0;
084             LeaveCriticalSection( &cs_CSIPUACore_SIPCoreStop );
085             threadHandle = ( HANDLE )_beginthreadex( NULL, 0, &CSIPUACore_backgroundProcess, NULL, 0, &threadID );
086             if( threadHandle==0 ){
087                     //SIPCore_THREAD_CREATE_FAILED
088                     echoEvent( SIPCore_THREAD_CREATE_FAILED );
089             }
090
091             //=============================================
092             //added by chyei
093             //
094             FreeConsole();
095             //_CrtDumpMemoryLeaks();
096
097             AllocConsole();
098             hFile = CreateFile("CONOUT$", GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL);
099
100             InitializeCriticalSection( &cs_CSIPUACore_tryToGetIPAddress );
101
102             ifANYAddress = false;
103             ifhostIPAddress = true;
104             tryToGetIPAddressHandle = NULL;
105             ipthreadID = 0;
106
107             IPChangedCallback.SetCallback(this, IpChangeHandler);
108             CAddressChange::AddIpChangeNotify(&IPChangedCallback);
109             CAddressChange::StartIpChangeNotify();
110             //=============================================
111         }
112
113         //=============================================
114         //added by chyei
115         //
116         //*************************************************************************
117         //*                                 Detect network switch events                                *
118         //*************************************************************************
119         //
120         // When network switching is too fast, that UA's IP is not real.
121
122         void CSIPUACore::IpChangeHandler(int Param)
123         {
124
125             //if(!noAddress)
126             //currentTime = performanceLog("DETECT_IP_MODIFICATION:");
127
128             EnterCriticalSection( &cs_CSIPUACore_tryToGetIPAddress);
129
```

```
130         currentTime = performanceLog2(NULL, NULL, 0);
131
132         int tmpPort;
133         int portBegin=0, portEnd=0, portDiff=1;
134         int size, stop;
135         char *tmpSourceIP, *tmpDestinationIP;
136         char *tmphostIPAddress;
137
138         tmphostIPAddress = ( char * )malloc( strlen(hostIPAddress) );
139         strcpy(tmphostIPAddress, hostIPAddress);
140
141
142         char *registra, *_registra;
143
144         registra = _strdup( registerInformation.registra );
145         for( ; ;registra++ )
146                 if( *registra==':' ){
147                         ++registra;
148                         break;
149                 }
150         _registra = _strdup( registra );
151
152
153         Sleep(20);
154
155
156         int retryTimes = 0;
157         for( stop=0, size=50; !stop; size*=2 )
158         {
159                 tmpSourceIP = ( char * )malloc( size );
160                 tmpDestinationIP = ( char * )malloc( size );
161
162                 switch( doIPSelectionWithDestination( _registra, tmpSourceIP, size-1, tmpDestinationIP, size-1 ) ){
163                         case 0:         //success...
164
165                                 sprintf(buf, "=====tmpSource: %s tmpDst: %s=====\n" , tmpSourceIP, tmpDestinationIP);
166                                 writeLogX(buf);
167
168                                 if( strcmp( tmpSourceIP, hostIPAddress ) )
169                                 {
170                                         if( hostIPAddress ) free( hostIPAddress );
171                                         hostIPAddress = _strdup( tmpSourceIP );
172                                         sprintf(buf, "=====doIPSelection:%s======\n" , hostIPAddress);
173                                         writeLogX(buf);
174                                 }
175                                 stop = 1;
176                                 break;
177                         case 2:         //i can't even malloc meomry of 'size' bytes?
178                         case 4:
179                                 writeLogX("=====switch point 2=====\n");
180                                 stop = 1;
181                                 break;
182                         case 3:
183                         case 5:
184                                 writeLogX("=====switch point 3=====\n");
185                                 break;
186                         case -1:
187                                 writeLogX("=====switch point 4=====\n");
188                                 if( hostIPAddress ) free( hostIPAddress );
189                                 hostIPAddress = NULL;
190                                 stop = 1;
191                                 break;
192                 }
193                 if( tmpSourceIP ) free( tmpSourceIP );
194                 if( tmpDestinationIP ) free( tmpDestinationIP );
195                 free( _registra );
196         }
197
198         if( _natTraversalType==AUTO_DETECT_NAT )
199                 _natTraversalType = detectNAT();
200         switch( _natTraversalType ){
201                 case NO_NAT:
202                         portBegin = 5060;
203                         portEnd = 5160;
204                         portDiff = 10;
205                         break;
```

```
206                     case USE_UPNP:
207                             for( tmpPort=5060; tmpPort<5160; tmpPort+=10 ){
208                                     if( unUseableHostPort( tmpPort ) )
209                                             continue;
210                                     if( doUPnPGetAPort( tmpPort, &natExternalSIPPort )==0 ){
211                                             portBegin = tmpPort;
212                                             portEnd = tmpPort;
213                                             portDiff = 1;
214                                             break;
215                                     }
216                             }
217                             doUPnPGetAPort( 9000, &rtpNatExternalAudioPort );
218                             break;
219                     case USE_STATIC_ASSIGN:
220                             portBegin = hostSIPPort;
221                             portEnd = hostSIPPort;
222                             portDiff = 1;
223                             break;
224             }
225
226             if(hostIPAddress)
227             {
228                     bool tmpflag;
229                     ADDRINFO *oldAddrInfo, *newAddrInfo;
230
231                     getaddrinfo(tmphostIPAddress, "5060", NULL, &oldAddrInfo);
232                     getaddrinfo(hostIPAddress, "5060", NULL, &newAddrInfo);
233
234                     if( _internetFamily==USE_IPV6 )
235                     {
236                             //writeLogX("=====IPv6 compar=====\n");
237                             if(!memcmp( oldAddrInfo->ai_addr, newAddrInfo->ai_addr, sizeof(struct sockaddr_in6)))
238                                     tmpflag = true;
239                             else
240                                     tmpflag = false;
241                     }
242                     else
243                     {
244                             if(!memcmp( oldAddrInfo->ai_addr, newAddrInfo->ai_addr, sizeof(struct sockaddr_in)))
245                                     tmpflag = true;
246                             else
247                                     tmpflag = false;
248                     }
249
250
251                     if(tmpflag)
252                     {
253                             writeLogX("=====ORG IP NOTIFY=====\n");
254                             sprintf(buf, "////// %f \\\\\\\\n", currentTime - previousTime);
255                             writeLogX(buf);
256     #ifdef IPV6_TEST
257                             if( currentTime - previousTime > 15.0)
258                             {
259                                     if(threadflag == 0)
260                                     {
261                                             threadflag = 1;
262                                             ifhostIPAddress = false;
263                                             checkAddress = 1;
264                                             if( !tryToGetIPAddressHandle )
265                                             {
266                                                     writeLogX("=====HANDLE START=====\n");
267                                                     tryToGetIPAddressHandle = ( HANDLE )_beginthreadex( NULL, 0,
&tryToGetIPAddress, NULL, 0, &ipthreadID );
268                                             }
269                                     }
270                             }
271                             else
272     #endif
273                             {
274                                     threadflag = 0;
275                                     ifhostIPAddress = true;
276                                     tryToGetIPAddressHandle = NULL;
277                                     echoEvent( SIPCore_IPCHANGE_ORGIP_NOTIFY );
278                             }
279                     }
280                     else
```

```
281                     {
282                             performanceLog2("DETECT_IP_MODIFICATION:", currentTime, 1);
283                             performanceLog("IP_CHANGED:");
284                             previousTime = performanceLog2("IP_CHANGED:", 0, 0);
285                             writeLogX("=====NEW IP NOTIFY=====\n");
286                             //sprintf(buf, "ip:%s\n", hostIPAddress);
287                             //writeLogX(buf);
288                             threadflag = 0;
289                             ifhostIPAddress = true;
290                             tryToGetIPAddressHandle = NULL;
291                             modifyeXosipIP(portBegin, portEnd, portDiff);
292                             echoEvent( SIPCore_IPCHANGE_NEWIP_NOTIFY );
293
294                             noAddress = false;
295                     }
296
297                     freeaddrinfo(oldAddrInfo);
298                     freeaddrinfo(newAddrInfo);
299
300             }
301             else
302             {
303                     writeLogX("=====NO IP NOTIFY=====\n");
304                     //performanceLog("DetectIP_NON");
305                     echoEvent( SIPCore_IPCHANGE_NOIP_NOTIFY );
306                     ifhostIPAddress = false;
307
308                     if( !tryToGetIPAddressHandle )
309                     {
310                             writeLogX("=====HANDLE START=====\n");
311                             tryToGetIPAddressHandle = ( HANDLE )_beginthreadex( NULL, 0, &tryToGetIPAddress, NULL, 0,
&ipthreadID );
312                     }
313
314                     hostIPAddress = ( char * )malloc( strlen(tmphostIPAddress) );
315                     strcpy(hostIPAddress, tmphostIPAddress);
316
317                     noAddress = true;
318
319             }
320             free(tmphostIPAddress);
321
322             LeaveCriticalSection( &cs_CSIPUACore_tryToGetIPAddress);
323
324         }
325
326         int CSIPUACore::modifyeXosipIP(int portBegin, int portEnd, int portDiff)
327         {
328             if(eXosip_lock() == 0)
329             {
330                     for(int tmpPort = portBegin; tmpPort<=portEnd; tmpPort+=portDiff)
331                     {
332                             //int ret;
333                             /*
334                             if(ifANYAddress)
335                             {
336                                     //writeLogX("Just Address\n");
337                                     if( eXosip_modify_ip(hostIPAddress, tmpPort, 0) < 0)
338                                             continue;
339                             }
340                             else
341                             {
342                                     if( eXosip_modify_ip(hostIPAddress, tmpPort, 1) < 0)
343                                             continue;
344                             }
345                             */
346
347
348                             if( eXosip_modify_ip(hostIPAddress, tmpPort, 2) < 0)
349                                     continue;
350
351
352                             if( _natTraversalType!=NO_NAT )
353                             {
354                                     eXosip_force_localip( natExternalIPAddress );
355                                     eXosip_set_firewallip( natExternalIPAddress );
```

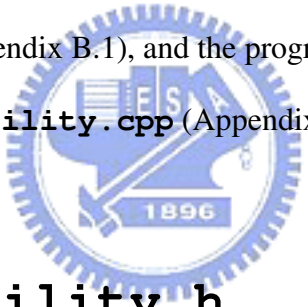
```
356                             eXosip_set_firewallsipPort( natExternalSIPPort );
357                     }
358
359                     hostSIPPort = sipPort = tmpPort;
360                     /*
361                     if(!ifANYAddress)
362                     {
363                             ifANYAddress = true;
364                             //Sleep(100);
365                     }
366                     */
367                     //Sleep(100);
368
369                     eXosip_unlock();
370                     return 1;
371             }
372             eXosip_unlock();
373         }
374         return -1;
375     }
376
377     int CSIPUACore::doREINVITE()
378     {
379         rtpLocalAudioPort = getRTPLocalAudioPort();
380         if( rtpLocalIP ) free( rtpLocalIP );
381         rtpLocalIP = ( char * )malloc( 50 );
382         if(eXosip_guess_localip( _internetFamily==USE_IPV4? PF_INET:PF_INET6, rtpLocalIP, 49 ) != 0)
383                 return -1;
384
385         //sprintf(buf, "rtplocalIP:%s rtpport:%d\n", rtpLocalIP, rtpLocalAudioPort);
386         //writeLogX(buf);
387         if(eXosip_lock() == 0)
388         {
389                 if(eXosip_off_hold_call(dialogID , rtpLocalIP , rtpLocalAudioPort) != 0)
390                 {
391                         eXosip_unlock();
392                         return -1;
393                 }
394                 eXosip_unlock();
395         }
396         else
397                 return -1;
398         return 0;
399     }
400
401     void CSIPUACore::dealRecvReInvite( const eXosip_event_t *eXosipEvent ) {
402
403         char *RTPPort=NULL;
404         osip_from_t *tmpCallerURI;
405
406         if( rtpRemoteIP ) free( rtpRemoteIP );
407                 rtpRemoteIP = NULL;
408         if( eXosipEvent->remote_sdp_audio_ip && strlen( eXosipEvent->remote_sdp_audio_ip )!=0 ){  //invite coming with
SDP
409                 rtpRemoteIP = _strdup( eXosipEvent->remote_sdp_audio_ip );
410                 rtpRemoteAudioPort = eXosipEvent->remote_sdp_audio_port;
411                 if( eXosipEvent->payload==-1 || eXosipEvent->payload_name==NULL ||
strlen( eXosipEvent->payload_name )==0 ){
412                         //no codec can support, error... send 415 response and terminate the call automatically
413                         echoEvent( SIPCore_RECV_INVITE_FAILED_NO_CODEC_SUPPORT );
414                         if( eXosip_lock()==0 ){
415                                 if( eXosip_answer_call( dialogID, 415, NULL )==-1 ){
416                                         ;//ignore the error ??   'SIPCore_RECV_INVITE_SEND_415_FAILED'
417                                 }
418                                 eXosip_unlock();
419
420                                 //a call terminate
421                                 aCallTerminatedVariablesReset();
422                                 return;
423                         }
424                 }
425                 else
426                         rtpRemotePayloadType = eXosipEvent->payload;
427         }
428
429         if( _subject ) free( _subject );
```

```
430         if( _callerURI ) free( _callerURI );
431         _subject = ( eXosipEvent->subject==NULL||strlen( eXosipEvent->subject )==0 )?
432                         _strdup( "General Call" ):_strdup( eXosipEvent->subject );
433         if( osip_from_init( &tmpCallerURI )!=-1 &&
434                 osip_from_parse( tmpCallerURI, eXosipEvent->remote_uri )!=-1 ){
435                 _callerURI = ( char * )malloc( strlen( eXosipEvent->remote_uri ) );
436                 sprintf( _callerURI, "%s:%s@%s", osip_uri_get_scheme( tmpCallerURI->url ),
437                                                         osip_uri_get_username( tmpCallerURI->url ),
438                                                         osip_uri_get_host( tmpCallerURI->url ) );
439         }
440         else
441                 _callerURI = _strdup( eXosipEvent->remote_uri );
442         osip_from_free( tmpCallerURI );
443
444         echoEvent(SIPCore_RECV_REINVITE);
445     }
446
447     unsigned __stdcall CSIPUACore::tryToGetIPAddress( void *argument )
448     {
449         while(checkAddress == 1)
450         {
451                 CSIPUACore_UACore->AppSIPMobilityGetAdaptersAddresses();
452                 Sleep(10);
453         }
454
455         while(!CSIPUACore_UACore->ifhostIPAddress)
456         {
457                 Sleep(1);
458                 CSIPUACore_UACore->IpChangeHandler(0);
459         }
460         //3X("=====END THREAD=====\n");
461         //_endthread();
462         return 0;
463     }
464
465     void CSIPUACore::AppSIPMobilityGetAdaptersAddresses()
466     {
467         PIP_ADAPTER_ADDRESSES AdapterAddresses = NULL;
468           ULONG OutBufferLength = 0;
469           ULONG RetVal = 0, i;
470           char addressString[128], portString[8];
471         int errorCode;
472
473         for (i = 0; i < 2; i++)
474         {
475                 RetVal = GetAdaptersAddresses(AF_INET6, 0, NULL, AdapterAddresses, &OutBufferLength);
476
477                 if (RetVal != ERROR_BUFFER_OVERFLOW)
478                 {
479                     break;
480                 }
481
482                 if (AdapterAddresses != NULL)
483                 {
484                     FREE(AdapterAddresses);
485                 }
486
487                 AdapterAddresses = (PIP_ADAPTER_ADDRESSES)MALLOC(OutBufferLength);
488                 if (AdapterAddresses == NULL)
489                 {
490                     RetVal = GetLastError();
491                     break;
492                 }
493           }
494
495           if (RetVal == NO_ERROR)
496         {
497             // If successful, output some information from the data we received
498                 PIP_ADAPTER_ADDRESSES AdapterList = AdapterAddresses;
499                 while (AdapterList)
500                 {
501                         //printf("\tFriendly name: %S\n", AdapterList->FriendlyName);
502                         //printf("\tDescription: %S\n", AdapterList->Description);
503
504                         PIP_ADAPTER_UNICAST_ADDRESS AddressList= AdapterList->FirstUnicastAddress;
505                         while(AddressList)
```

```
506                         {
507                                 //fprintf(stdout, "Address List GET IN\n");
508                                 //printf("\tFriendly name: %S\n", AdapterAddresses->FriendlyName);
509
510                                 if( ( errorCode=getnameinfo( AddressList->Address.lpSockaddr, sizeof( struct sockaddr_in6 ),
511                                                 addressString, 127,
512                                                 portString, 7,
513                                                 NI_NUMERICHOST|NI_NUMERICSERV ) )!=0 )
514                                 {
515                                         //sprintf( stringBuffer, "===== Error happened in getnameinfo()
AppSIPMobilityDstAddrSelection in IPv4 with errorCode %d =====\n\n", errorCode );
516                                         //OutputConsoleString(stringBuffer);
517                                         AddressList = AddressList->Next;
518                                         continue;
519                                 }
520                                 if(AddressList->DadState == 1)
521                                 {
522                                         if(checkAddress == 1)
523                                         {
524                                                 performanceLog("GET_IPV6_ADDRESS:");
525                                                 checkAddress = 0;
526                                         }
527                                         //sprintf( stringBuffer, "\t%s\n\n", addressString );
528                                         //OutputConsoleString(stringBuffer);
529                                         //fprintf(stdout, "Unicast Addr DAD Type: %d\n", AddressList->DadState);
530                                 }
531                                 AddressList = AddressList->Next;
532                         }
533
534
535                         AdapterList = AdapterList->Next;
536                     }
537                 }
538                 else {
539                   LPVOID MsgBuf;
540
541                   printf("Call to GetAdaptersAddresses failed.\n");
542                   if (FormatMessage(
543                     FORMAT_MESSAGE_ALLOCATE_BUFFER |
544                     FORMAT_MESSAGE_FROM_SYSTEM |
545                     FORMAT_MESSAGE_IGNORE_INSERTS,
546                     NULL,
547                     RetVal,
548                     MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
549                     (LPTSTR) &MsgBuf,
550                     0,
551                     NULL )) {
552                     printf("\tError: %s", MsgBuf);
553                   }
554                   LocalFree(MsgBuf);
555                 }
556
557                 if (AdapterAddresses != NULL) {
558                     FREE(AdapterAddresses);
559                 }
560
561             return;
562         }
563
564         //=============================================================
```

Appendix B
The SIP Mobility Module Program in the RADVISION Implementation

Appendix B lists the source code of the SIP Mobility module program in the RADVISION implementation. The SIP Mobility module is implemented as a C++ program, which consists of the header files **`AddressChange.h`** (Appendix A.1), **`NTP_sipmobility.h`** (Appendix B.1), and the program files **`AddressChange.h`** (Appendix A.1), **`NTP_sipmobility.cpp`** (Appendix B.2).



B.1 `NTP_sipmobility.h`

```
01      #ifndef _NTP_sipmobility_H_
02      #define _NTP_sipmobility_H_
03
04      #ifdef __cplusplus
05      extern "C" {
06      #endif
07          double PerformanceTimeLog(char *string, double value, int ifprint);
08          RvStatus AppSIPMobilityInit();
09          RvStatus AppSIPMobilityQuit();
10          RvStatus AppSIPMobilityAddCallIndex(AppCall *callLegPointer);
11          RvStatus AppSIPMobilityRemoveCallIndex(AppCall *callLegPointer);
12      #ifdef __cplusplus
13      };
14      #endif
15
16      #endif
```

B.2 NTP_sipmobility.cpp

```
0001
0002    #include "AddressChange.h"
0003    using namespace ADDRESSCHANGE;
0004
0005    #include <ws2tcpip.h>
0006
0007    #include "NTP_call.h"
0008    #include "NTP_reg.h"
0009    #include "NTP_tclGen.h"
0010    #include "NTP_rtp.h"
0011    #include "NTP_mgr.h"
0012    #include "NTP_sipmobility.h"
0013    //#include "CH_utils.h"
0014
0015    #define PORT "5060"
0016
0017    //#ifndef SIP_M_TEST
0018    //#define SIP_M_TEST
0019    #undef SIP_M_TEST
0020
0021    //extern TestAppMgr *g_pMgr;
0022
0023    void AppSIPMobilityIpChangeHandler(void *pri);
0024
0025    RvStatus AppSIPMobilityIpChangeHandlerFunction( AppCall *currentCall );
0026
0027    RvStatus AppSIPMobilityDstAddrSelection( const char *remoteAddr,
0028                                             struct sockaddr_storage *dstIPv4Addr,
0029                                             struct sockaddr_storage *dstIPv6Addr,
0030                                             int *dstIPv4AddrNum, int *dstIPv6AddrNum);
0031
0032    RvStatus AppSIPMobilitySrcAddrSelectionWithDst(struct sockaddr_storage *dstIPv4Addr,
0033                                                   struct sockaddr_storage *dstIPv6Addr,
0034                                                   struct sockaddr_storage *srcIPv4Addr,
0035                                                   struct sockaddr_storage *srcIPv6Addr);
0036
0037
0038    RvStatus AppSIPMobilityGetDstAddress(const RvChar *dstAddress, char *retAddress);
0039
0040
0041    RvStatus AppSIPMobilityGetContactHeaderHostName(AppCall *currentCall, RvBool isLocal, char *buf, RvUint32
*len);
0042
0043    RvStatus checkAddressAvalibility(sockaddr_storage *srcIPv6Addr, sockaddr_storage *dstIPv6Addr);
0044    RvStatus AppSIPMobilityAddNewCall();
0045    RvStatus AppSIPMobilityListAddrQuery(char *localIPv4Addr, char *localIPv6Addr, bool *ipv4Exist, bool *ipv6Exist);
0046
0047
0048    static std::vector<AppCall*> AppCallList;
0049    static RvBool                s_WSAON;
0050
0051    CRITICAL_SECTION m_IPHandlerSection;
0052
0053    double currentTime = 0, previousTime = 1000000;
0054    int checkAddress = 0;
0055    int checkAddressCounter = 0;
0056    int pre_addrTypePrefer = 1;
0057    int address_set = 0;
0058    int reinviteCounter = 0;
0059    int resetflag = 0;
0060    char stringBuffer[200];
0061    int listLock = 0;
0062    int loopflag = 1;
0063
0064    double PerformanceTimeLog(char *item, double value, int ifprint)
```

```
0065    {
0066      char buffer[99];
0067
0068      LARGE_INTEGER s, g_ptFreq;
0069      double currentTime;
0070
0071      if(value == NULL || value == -1)
0072      {
0073              QueryPerformanceFrequency(&g_ptFreq);
0074              QueryPerformanceCounter(&s);
0075              currentTime = (double)s.QuadPart / (double)g_ptFreq.QuadPart;
0076      }
0077      else
0078      {
0079              currentTime = value;
0080      }
0081
0082      sprintf(buffer, "%s:\t%f\n", item, currentTime);
0083
0084      if(ifprint == 1)
0085              OutputDebugString(buffer);
0086
0087
0088      return currentTime;
0089    }
0090
0091
0092    void OutputConsoleString(char *string)
0093    {
0094      fprintf(stderr, "%s", string);
0095    }
0096
0097
0098    RvStatus AppSIPMobilityInit()
0099    {
0100      CallbackEntry cbkent;
0101      cbkent.func = AppSIPMobilityIpChangeHandler;
0102      cbkent.pri = NULL;
0103      CAddressChange::AddIpChangeNotify(cbkent);
0104      CAddressChange::StartIpChangeNotify();
0105
0106      s_WSAON = RV_FALSE;
0107
0108      InitializeCriticalSection(&m_IPHandlerSection);
0109      //AppSIPMobilityAddNewCall();
0110
0111      return RV_OK;
0112
0113    }
0114
0115    RvStatus AppSIPMobilityQuit()
0116    {
0117      int errorCode;
0118
0119      if(s_WSAON == RV_TRUE)
0120      {
0121              if( ( errorCode=WSACleanup() )!=0 )
0122              {
0123                      sprintf( stringBuffer, "Error happened in WSACleanup()... with errorCode %d\n", errorCode );
0124                      OutputConsoleString(stringBuffer);
0125                      return 1;
0126              }
0127              s_WSAON = RV_FALSE;
0128      }
0129
0130      DeleteCriticalSection(&m_IPHandlerSection);
0131
0132      return RV_OK;
0133    }
0134
0135    RvStatus AppSIPMobilityAddCallIndex(AppCall *callLegPointer)
0136    {
0137      AppCallList.push_back(callLegPointer);
0138      return RV_OK;
0139    }
0140
```

```
0141    RvStatus AppSIPMobilityRemoveCallIndex(AppCall *callLegPointer)
0142    {
0143      while (listLock == 1)
0144      {
0145              fprintf(stdout, "list Locked at Remove\n");
0146              Sleep(5);
0147      }
0148      for(vector<AppCall *>::iterator it = AppCallList.begin(); it != AppCallList.end() ; it++)
0149      {
0150              listLock = 1;
0151              if(*it == callLegPointer)
0152              {
0153                      AppCallList.erase(it);
0154                      break;
0155              }
0156      }
0157      listLock = 0;
0158      return RV_OK;
0159    }
0160    //=============================================
0161    //added by chyei
0162    //
0163    //*************************************************************************
0164    //*                                Detect network switch events                                *
0165    //*************************************************************************
0166    //
0167    // When network switching is too fast, that UA's IP is not real.
0168    /*
0169
0170      //if we are in the IPv4 only, the remote should be IPv4 too.
0171      //compare current ip address and new address
0172
0173      //if we are in the IPv6 only, the remote should be IPv6 too.
0174      //compare current ip address and new address
0175
0176      //how about dual stack?
0177      //use previous address family, if not avalible, change address family
0178
0179
0180       //after get the remote address , we should get the local connection address
0181      //check the address list and compare with the information in callleg
0182      //AppSIPMobilitySrcAddrSelectionWithDst(&dstIPv4Addr, &dstIPv6Addr, &srcIPv4Addr, &srcIPv6Addr);
0183
0184
0185      //we should register new address if modified
0186
0187
0188    */
0189
0190
0191
0192    void AppSIPMobilityIpChangeHandler(void *pri)
0193    {
0194
0195      int reConnected = 0;
0196      int ifConnected = 0;
0197      AppCall *tmpCall;
0198      double timePoint = PerformanceTimeLog("DETECT_IP_MODIFICATION", NULL, false);
0199
0200
0201      //currentTime =
0202      checkAddress = 0;
0203      checkAddressCounter = 0;
0204
0205      //check all call status
0206      ///compare the address in use (SDP? & local contact?)
0207      //if status == session-ing
0208      //send re-Invite
0209
0210      while(listLock == 1)
0211      {
0212              fprintf(stdout, "list Locked at FOR LOOP\n");
0213              Sleep(5);
0214      }
0215
0216      EnterCriticalSection(&m_IPHandlerSection);
```

```
0217        vector<AppCall *>::iterator it;
0218        for( it = AppCallList.begin(); it != AppCallList.end() ; it++)
0219        {
0220                listLock    = 1;
0221                ifConnected = 1;
0222                AppCall *currentCall = (*it);
0223
0224
0225
0226                if( currentCall->eState == RVSIP_CALL_LEG_STATE_CONNECTED )
0227                {
0228                        RvStatus rv;
0229                        rv = AppSIPMobilityIpChangeHandlerFunction(currentCall);
0230
0231                        //fprintf(stdout, "Time: %f\n", timePoint - previousTime);
0232                        //if(timePoint - previousTime > 12.0)
0233                        if(timePoint - previousTime > 12.0 && pre_addrTypePrefer == 0)
0234                        {
0235                                fprintf(stdout, "GET IN\n");
0236                                checkAddress = 1;
0237                                while(rv != RV_OK)
0238                                {
0239                                        checkAddressCounter++;
0240                                        Sleep(10);
0241                                        rv = AppSIPMobilityIpChangeHandlerFunction(currentCall);
0242                                }
0243                        }
0244
0245                        fprintf(stdout , "Counter: %d\n\n", reinviteCounter);
0246                        //OutputConsoleString(stringBuffer);
0247
0248                        if( ( ( (reinviteCounter+1)%50 == 0 ) || ((reinviteCounter+1)%50 == 1)) && pre_addrTypePrefer == 0 )
0249                        //if( (reinviteCounter+1)%100 == 0 )
0250                        {
0251                                tmpCall = currentCall;
0252                                reConnected = 1;
0253                                reinviteCounter++;
0254                        }
0255                }
0256                else
0257                {
0258                        tmpCall = currentCall;
0259                        ifConnected = 0;
0260                        fprintf(stdout, "Fail\n");
0261                }
0262        }
0263
0264        listLock = 0;
0265
0266        if(reConnected)
0267        {
0268                fprintf(stdout, "Reconnect\n");
0269                Sleep(1000);
0270                AppCallDisconnect(tmpCall);
0271                Sleep(500);
0272                AppSIPMobilityAddNewCall();
0273        }
0274
0275        if( ifConnected == 0)
0276        {
0277                Sleep(500);
0278                AppSIPMobilityAddNewCall();
0279        }
0280
0281        LeaveCriticalSection(&m_IPHandlerSection);
0282      }
0283
0284
0285      RvStatus AppSIPMobilityIpChangeHandlerFunction( AppCall *currentCall )
0286      {
0287        struct sockaddr_storage dstIPv4Addr, dstIPv6Addr;
0288        struct sockaddr_storage srcIPv4Addr, srcIPv6Addr;
0289
0290        RvChar localIPv4Address[APP_MGR_ADDR_STRING_SIZE]= {'\0'};
0291        RvChar localIPv6Address[APP_MGR_ADDR_STRING_SIZE]= {'\0'};
0292        RvChar srcIPv4AddrString[APP_MGR_ADDR_STRING_SIZE]= {'\0'};
```

```
0293        RvChar srcIPv6AddrString[APP_MGR_ADDR_STRING_SIZE]= {'\0'};
0294        RvChar remoteContactAddr[APP_MGR_ADDR_STRING_SIZE]= {'\0'};
0295        RvChar remoteAddr[APP_MGR_ADDR_STRING_SIZE]= {'\0'};
0296
0297        RvStatus rv;
0298        RvUint16 port = 0;
0299        RvUint32 tmplen = 0;
0300
0301        char portString[7];
0302        char tmpHostAddress[APP_MGR_ADDR_STRING_SIZE];
0303        bool exitSrcIPv4 = false, exitSrcIPv6 = false;
0304        bool ipv4Exist = false, ipv6Exist = false;
0305        int     dstIPv4AddrNum = 0, dstIPv6AddrNum = 0;
0306        int addrTypePrefer = 0;
0307        int errorCode;
0308        //double detect_ip_time = 0.0;
0309
0310        memset( &dstIPv4Addr, 0, sizeof( struct sockaddr_storage ) );
0311        memset( &dstIPv6Addr, 0, sizeof( struct sockaddr_storage ) );
0312        memset( &srcIPv4Addr, 0, sizeof( struct sockaddr_storage ) );
0313        memset( &srcIPv6Addr, 0, sizeof( struct sockaddr_storage ) );
0314
0315        sprintf( stringBuffer, "================= IP Change Handler Start ==================\n" );
0316        //EnterCriticalSection(&m_IPHandlerSection);
0317
0318        currentTime = PerformanceTimeLog("DETECT_IP_MODIFICATION", NULL, 0);
0319
0320        if(currentCall->addrTypePrefer == 0)
0321        {
0322                pre_addrTypePrefer = 0;
0323                addrTypePrefer = 0;
0324      #ifdef SIP_M_TEST
0325                        addrTypePrefer = 1;
0326      #endif
0327          }
0328          else
0329          {
0330                pre_addrTypePrefer = 1;
0331                addrTypePrefer = 1;
0332      #ifdef SIP_M_TEST
0333                        addrTypePrefer = 0;
0334      #endif
0335          }
0336
0337          /*** Get RADVISION Local Address ***/
0338          rv = AppCallGetLocalAddress(currentCall, RVSIP_TRANSPORT_UDP,
RVSIP_TRANSPORT_ADDRESS_TYPE_IP, localIPv4Address, &port);
0339          if(rv != RV_OK)
0340          {
0341                sprintf( stringBuffer, "===== NO IPv4 Address =====\n" );
0342                OutputConsoleString(stringBuffer);
0343          }
0344          else
0345          {
0346          sprintf( stringBuffer, "< RADVISION AppCallGetLocalAddress IPv4 address >\n\t%s\n\n", localIPv4Address );
0347                OutputConsoleString(stringBuffer);
0348          }
0349
0350          rv = AppCallGetLocalAddress(currentCall, RVSIP_TRANSPORT_UDP,
RVSIP_TRANSPORT_ADDRESS_TYPE_IP6, tmpHostAddress, &port);
0351          if(rv != RV_OK)
0352          {
0353                sprintf( stringBuffer, "===== NO IPv6 Address =====\n" );
0354                OutputConsoleString(stringBuffer);
0355          }
0356          else
0357          {
0358                sprintf( stringBuffer, "< RADVISION AppCallGetLocalAddress IPv6 address >\n\t%s\n\n", tmpHostAddress );
0359                OutputConsoleString(stringBuffer);
0360          }
0361
0362          if(tmpHostAddress[0] == '[')    //IPv6 address
0363          {
0364                char *pointerS, *pointerE;
0365                pointerS = strchr(tmpHostAddress, '[');
0366                pointerE = strrchr(tmpHostAddress, ']');
```

```
0367                    if(pointerS != NULL && pointerE != NULL)
0368                    {
0369                            strncpy(localIPv6Address, (pointerS+1), (pointerE-pointerS-1));
0370                    }
0371            }
0372            else
0373                    strcpy(localIPv6Address, tmpHostAddress);
0374
0375            /*** Check the RADVISION address if still exist ***/
0376            AppSIPMobilityListAddrQuery(localIPv4Address, localIPv6Address, &ipv4Exist, &ipv6Exist);
0377
0378
0379            /*** Get Remote Contact ***/
0380            AppSIPMobilityGetContactHeaderHostName(currentCall, RV_FALSE, remoteContactAddr, &tmplen);
0381            sprintf( stringBuffer, "< Remote Contact Address >\n\t%s\n\n", remoteContactAddr );
0382            OutputConsoleString(stringBuffer);
0383
0384            Sleep(5);
0385
0386            AppSIPMobilityDstAddrSelection(remoteContactAddr, &dstIPv4Addr, &dstIPv6Addr, &dstIPv4AddrNum,
&dstIPv6AddrNum);
0387
0388            if(dstIPv4AddrNum > 0)
0389            {
0390                    rv = AppSIPMobilitySrcAddrSelectionWithDst(&dstIPv4Addr, NULL, &srcIPv4Addr, NULL);
0391                    if(rv == RV_OK)
0392                    {
0393                            exitSrcIPv4 = true;
0394                            if( ( errorCode=getnameinfo( (struct sockaddr *) &srcIPv4Addr, sizeof( struct sockaddr_in ),
0395                                    srcIPv4AddrString, APP_MGR_ADDR_STRING_SIZE,
0396                                    portString, 7,
0397                                    NI_NUMERICHOST|NI_NUMERICSERV ) )!=0 )
0398                            {
0399            sprintf( stringBuffer, "===== Error happened in getnameinfo() AppSIPMobilitySrcAddrSelectionWithDst IPv4... with
errorCode %d =====\n\n", errorCode );
0400                                    OutputConsoleString(stringBuffer);
0401                                    exitSrcIPv4 = false;
0402                            }
0403                    }
0404            }
0405
0406            if(dstIPv6AddrNum > 0)
0407            {
0408                    rv = AppSIPMobilitySrcAddrSelectionWithDst(NULL, &dstIPv6Addr, NULL, &srcIPv6Addr);
0409                    if(rv == RV_OK)
0410                    {
0411                            exitSrcIPv6 = true;
0412                            if( ( errorCode=getnameinfo( (struct sockaddr *) &srcIPv6Addr, sizeof( struct sockaddr_in6 ),
0413                                            srcIPv6AddrString, APP_MGR_ADDR_STRING_SIZE,
0414                                            portString, 7,
0415                                            NI_NUMERICHOST|NI_NUMERICSERV ) )!=0 )
0416                            {
0417            sprintf( stringBuffer, "===== Error happened in getnameinfo() AppSIPMobilitySrcAddrSelectionWithDst IPv6... with
errorCode %d =====\n\n", errorCode );
0418                                    OutputConsoleString(stringBuffer);
0419                                    exitSrcIPv6 = false;
0420                            }
0421
0422                            if( strncmp(srcIPv6AddrString, "2001:", 5) != 0 )
0423                            {
0424                                    sprintf( stringBuffer, "===== Not an unicast IPv6 address =====\n\n");
0425                                    OutputConsoleString(stringBuffer);
0426                                    exitSrcIPv6 = false;
0427
0428            //#ifdef SIP_M_TEST
0429                                    if(strncmp(srcIPv6AddrString, "fe80:", 5) == 0 )
0430                                    {
0431                                            if(checkAddress == 1)
0432                                            {
0433                                                    checkAddress = 0;
0434                                                    PerformanceTimeLog("GET_IPV6_ADDRESS", NULL, 1);
0435                                                    //return RV_OK;
0436                                            }
0437                                    }
0438
0439                                    if(addrTypePrefer == 1)
```

```
0440                                    {
0441                                            //LeaveCriticalSection(&m_IPHandlerSection);
0442                                            return RV_ERROR_UNKNOWN;
0443                                    }
0444            //#endif
0445                            }
0446                    }
0447            }
0448
0449            #ifdef SIP_M_TEST
0450              if(addrTypePrefer == 0)
0451              {
0452                    if( !(dstIPv4AddrNum > 0 && exitSrcIPv4) )
0453                    {
0454                            //LeaveCriticalSection(&m_IPHandlerSection);
0455                            return RV_ERROR_UNKNOWN;
0456                    }
0457              }
0458              else
0459              {
0460                    if( ! (dstIPv6AddrNum > 0 && exitSrcIPv6) )
0461                    {
0462                            //LeaveCriticalSection(&m_IPHandlerSection);
0463                            return RV_ERROR_UNKNOWN;
0464                    }
0465              }
0466
0467            #else
0468              if(addrTypePrefer == 0)
0469              {
0470                    if(dstIPv4AddrNum > 0 && exitSrcIPv4)
0471                            addrTypePrefer = 0;
0472                    else if(dstIPv6AddrNum > 0 && exitSrcIPv6)
0473                            addrTypePrefer = 1;
0474                    else
0475                    {
0476                            //LeaveCriticalSection(&m_IPHandlerSection);
0477                            return RV_ERROR_UNKNOWN; //error
0478                    }
0479              }
0480              else
0481              {
0482                    if(dstIPv6AddrNum > 0 && exitSrcIPv6)
0483                            addrTypePrefer = 1;
0484                    else if(dstIPv4AddrNum > 0 && exitSrcIPv4)
0485                            addrTypePrefer = 0;
0486                    else
0487                    {
0488                            //LeaveCriticalSection(&m_IPHandlerSection);
0489                            return RV_ERROR_UNKNOWN; //error
0490                    }
0491              }
0492            #endif
0493
0494              if(addrTypePrefer == 0)
0495              {
0496                    sprintf( stringBuffer, "< Current Address Type Prefer => IPv4 >\n\n");
0497                    OutputConsoleString(stringBuffer);
0498              }
0499              else
0500              {
0501                    sprintf( stringBuffer, "< Current Address Type Prefer => IPv6 >\n\n");
0502                    OutputConsoleString(stringBuffer);
0503              }
0504
0505              if(addrTypePrefer == 0)
0506              {
0507
0508                    if(ipv4Exist)
0509                    {
0510                            //original exist
0511                            sprintf( stringBuffer, "<< IPv4 address exist, This is the original IPv4 address... Do Nothing >>\n" );
0512                            OutputConsoleString(stringBuffer);
0513                            return RV_OK;
0514                    }
0515                    else if(strcmp(localIPv4Address, "0.0.0.0") == 0)
```

```
0516                {
0517
0518                }
0519        #ifdef SIP_M_TEST
0520                else if(strcmp(srcIPv4AddrString, "140.113.131.70") == 0)
0521                {
0522                        sprintf( stringBuffer, "<< We filter the LAN ipv4 address ... Do Nothing >>\n" );
0523                        OutputConsoleString(stringBuffer);
0524                        //LeaveCriticalSection(&m_IPHandlerSection);
0525                        return RV_ERROR_UNKNOWN;
0526                }
0527        #endif
0528                else //specify address
0529                {
0530                        if(strcmp(srcIPv4AddrString, localIPv4Address) == 0 )
0531                        {
0532                                if(pre_addrTypePrefer == 0)
0533                                {
0534                                        sprintf( stringBuffer, "<< This is the original IPv4 address... Do Nothing >>\n" );
0535                                        OutputConsoleString(stringBuffer);
0536                                        //LeaveCriticalSection(&m_IPHandlerSection);
0537                                        return RV_ERROR_UNKNOWN;
0538                                }
0539                                else
0540                                {
0541                                        PerformanceTimeLog("DETECT_IP_MODIFICATION", currentTime, 1);
0542                                        PerformanceTimeLog("IP_CHANGED", NULL, true);
0543                                        //RTP_TestClosePlayAudio();
0544          AppCallSetNewRtpAddr(currentCall, srcIPv4AddrString, RVSIP_TRANSPORT_ADDRESS_TYPE_IP);
0545
0546                                        TclSetVariable(REINVIE_SETTING_DNS_PREFER_ADDRESS_TYPE, "IPv4");
0547                                        TclSetVariable(REINVIE_SETTING_LOCAL_CONTACT_ADDRESS, "");
0548                                        TclSetVariable(REINVIE_SETTING_REMOTE_CONTACT_ADDRESS, "");
0549
0550                                        Sleep(35);
0551                                        rv = AppCallAppCallPrepareAndModifyv2(currentCall);
0552                                        if(rv != RV_OK)
0553                                        {
0554                                                //re-INVITE Error
0555                                                resetflag = 1;
0556                                        }
0557                                        previousTime = PerformanceTimeLog("REINVITE_IPV4", NULL, 1);
0558                                        sprintf( stringBuffer, "< RADVISION Send Re-Invite IPv4 SAME>\n");
0559                                        OutputConsoleString(stringBuffer);
0560                                        reinviteCounter++;
0561                                        pre_addrTypePrefer = 0;
0562                                }
0563                        }
0564                        else
0565                        {
0566                                PerformanceTimeLog("DETECT_IP_MODIFICATION", currentTime, 1);
0567                                PerformanceTimeLog("IP_CHANGED", NULL, 1);
0568                                //RTP_TestClosePlayAudio();
0569          AppCallSetNewRtpAddr(currentCall, srcIPv4AddrString, RVSIP_TRANSPORT_ADDRESS_TYPE_IP);
0570
0571          AppCallAddLocalAddr(srcIPv4AddrString, 5060, RVSIP_TRANSPORT_ADDRESS_TYPE_IP);
0572          sprintf( stringBuffer, "< RADVISION AddLocalAddress IPv4 address >\n\t%s\n\n", srcIPv4AddrString );
0573                                OutputConsoleString(stringBuffer);
0574
0575          AppCallSetLocalAddress(currentCall, RVSIP_TRANSPORT_UDP, RVSIP_TRANSPORT_ADDRESS_TYPE_IP,
srcIPv4AddrString, 5060);
0576          sprintf( stringBuffer, "< RADVISION SetLocalAddress IPv4 address >\n\t%s\n\n", srcIPv4AddrString );
0577                                OutputConsoleString(stringBuffer);
0578
0579                                TclSetVariable(REINVIE_SETTING_DNS_PREFER_ADDRESS_TYPE, "IPv4");
0580                                TclSetVariable(REINVIE_SETTING_LOCAL_CONTACT_ADDRESS, "");
0581
0582                                Sleep(35);
0583                                rv = AppCallAppCallPrepareAndModifyv2(currentCall);
0584                                if(rv != RV_OK)
0585                                {
0586                                        //re-INVITE Error
0587                                        resetflag = 1;
0588                                }
0589                                previousTime = PerformanceTimeLog("REINVITE_IPV4", NULL, 1);
0590                                sprintf( stringBuffer, "< RADVISION Send Re-Invite IPv4 >\n");
```

```
0591                                OutputConsoleString(stringBuffer);
0592                                reinviteCounter++;
0593                                pre_addrTypePrefer = 0;
0594        AppCallRemoveLocalAddr(localIPv4Address, 5060, RVSIP_TRANSPORT_ADDRESS_TYPE_IP);
0595                        }
0596                }
0597        }
0598
0599        if(addrTypePrefer == 1)
0600        {
0601                if(ipv6Exist)
0602                {
0603                        sprintf( stringBuffer, "<< IPv6 address exist, This is the original IPv6 address... Do Nothing >>\n" );
0604                        OutputConsoleString(stringBuffer);
0605                        return RV_OK;
0606                }
0607                else if(strcmp(localIPv6Address, "[::]%0") == 0)
0608                {
0609                        sprintf( stringBuffer, "<< IPv6 ANY Address... >>\n" );
0610                        OutputConsoleString(stringBuffer);
0611                }
0612        #ifdef SIP_M_TEST
0613                else if(strcmp(srcIPv6AddrString, "2001:238:202:0:cc10:6f7b:d13f:d539") == 0)
0614                {
0615                        sprintf( stringBuffer, "<< We filter the wlan ipv6 address ... Do Nothing >>\n" );
0616                        OutputConsoleString(stringBuffer);
0617                        //LeaveCriticalSection(&m_IPHandlerSection);
0618                        return RV_ERROR_UNKNOWN;
0619                }
0620        #endif
0621                else
0622                {
0623                        if(strcmp(srcIPv6AddrString, localIPv6Address) == 0 )
0624                        {
0625                                if( pre_addrTypePrefer == 1 )
0626                                {
0627                                        sprintf( stringBuffer, "< This is the original IPv6 address... Do Nothing >\n" );
0628                                        OutputConsoleString(stringBuffer);
0629                                        //LeaveCriticalSection(&m_IPHandlerSection);
0630                                        return RV_ERROR_UNKNOWN;
0631                                }
0632                                else
0633                                {
0634                                        char tmpString[APP_MGR_ADDR_STRING_SIZE];
0635                                        sprintf(tmpString, "[%s]", srcIPv6AddrString);
0636
0637                                        PerformanceTimeLog("DETECT_IP_MODIFICATION", currentTime, 1);
0638                                        PerformanceTimeLog("IP_CHANGED", NULL, 1);
0639                                        //RTP_TestClosePlayAudio();
0640          AppCallSetNewRtpAddr(currentCall, srcIPv6AddrString, RVSIP_TRANSPORT_ADDRESS_TYPE_IP6);
0641
0642                                        TclSetVariable(REINVIE_SETTING_DNS_PREFER_ADDRESS_TYPE, "IPv6");
0643                                        TclSetVariable(REINVIE_SETTING_LOCAL_CONTACT_ADDRESS, "");
0644                                        TclSetVariable(REINVIE_SETTING_REMOTE_CONTACT_ADDRESS, "");
0645
0646                                        Sleep(35);
0647                                        rv = AppCallAppCallPrepareAndModifyv2(currentCall);
0648                                        if(rv != RV_OK)
0649                                        {
0650                                                //re-INVITE Error
0651                                                resetflag = 1;
0652                                        }
0653                                        previousTime = PerformanceTimeLog("REINVITE_IPV6", NULL, 1);
0654                                        sprintf( stringBuffer, "< RADVISION Send Re-Invite IPv6 >\n");
0655                                        OutputConsoleString(stringBuffer);
0656                                        reinviteCounter++;
0657                                        pre_addrTypePrefer = 1;
0658                                }
0659                        }
0660                        else
0661                        {
0662                                char tmpString[APP_MGR_ADDR_STRING_SIZE];
0663                                char *pointerS;
0664                                if( (pointerS = strchr(srcIPv6AddrString, '%')) != NULL )
0665                                {
0666
```

```
0667                                    sprintf(tmpString, "[");
0668                                    strncat(tmpString + 1, srcIPv6AddrString, (pointerS - srcIPv6AddrString));
0669                                    strcat(tmpString, "]");
0670                                    strcat(tmpString, pointerS);
0671                            }
0672                            else
0673                                    sprintf(tmpString, "[%s]", srcIPv6AddrString);
0674
0675
0676                            PerformanceTimeLog("DETECT_IP_MODIFICATION", currentTime, 1);
0677                            PerformanceTimeLog("IP_CHANGED", NULL, 1);
0678                            //RTP_TestClosePlayAudio();
0679          AppCallSetNewRtpAddr(currentCall, srcIPv6AddrString, RVSIP_TRANSPORT_ADDRESS_TYPE_IP6);
0680
0681
0682          AppCallAddLocalAddr(srcIPv6AddrString, 5060, RVSIP_TRANSPORT_ADDRESS_TYPE_IP6);
0683                            sprintf( stringBuffer, "< RADVISION AddLocalAddress IPv6 address >\n\t%s\n\n", tmpString );
0684                            OutputConsoleString(stringBuffer);
0685
0686          AppCallSetLocalAddress(currentCall, RVSIP_TRANSPORT_UDP, RVSIP_TRANSPORT_ADDRESS_TYPE_IP6,
tmpString, 5060);
0687                            sprintf( stringBuffer, "< RADVISION SetLocalAddress IPv6 address >\n\t%s\n\n", tmpString );
0688                            OutputConsoleString(stringBuffer);
0689
0690                            TclSetVariable(REINVIE_SETTING_DNS_PREFER_ADDRESS_TYPE, "IPv6");
0691                            TclSetVariable(REINVIE_SETTING_LOCAL_CONTACT_ADDRESS, "");
0692                            TclSetVariable(REINVIE_SETTING_REMOTE_CONTACT_ADDRESS, "");
0693
0694                            Sleep(35);
0695                            rv = AppCallAppCallPrepareAndModifyv2(currentCall);
0696                            if(rv != RV_OK)
0697                            {
0698                                    //re-INVITE Error
0699                                    resetflag = 1;
0700                            }
0701                            previousTime = PerformanceTimeLog("REINVITE_IPV6", NULL, 1);
0702                            sprintf( stringBuffer, "< RADVISION Send Re-Invite IPv6 >\n");
0703                            OutputConsoleString(stringBuffer);
0704                            reinviteCounter++;
0705                            pre_addrTypePrefer = 1;
0706
0707          AppCallRemoveLocalAddr(tmpHostAddress, 5060, RVSIP_TRANSPORT_ADDRESS_TYPE_IP6);
0708
0709                    }
0710            }
0711          }
0712
0713          //LeaveCriticalSection(&m_IPHandlerSection);
0714          sprintf( stringBuffer, "====================== IP Change Handler End ======================\n\n" );
0715          OutputConsoleString(stringBuffer);
0716          return RV_OK;
0717        }
0718
0719
0720        RvStatus AppSIPMobilitySrcAddrSelectionWithDst(struct sockaddr_storage *dstIPv4Addr,
0721                                                                                        struct sockaddr_storage *dstIPv6Addr,
0722                                                                                        struct sockaddr_storage *srcIPv4Addr,
0723                                                                                        struct sockaddr_storage *srcIPv6Addr)
0724        {
0725          int errorcode;
0726          char srcIPv4AddrString[1024], srcIPv6AddrString[1024];
0727          unsigned long ipv4len, ipv6len;
0728
0729          struct sockaddr_storage addr;
0730
0731          if( dstIPv4Addr != NULL)
0732          {
0733                  SOCKET tmpsocketv4;
0734
0735                  if(dstIPv4Addr->ss_family != AF_INET)
0736                          return RV_ERROR_UNKNOWN;
0737
0738                  tmpsocketv4 = (int)socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
0739                  if(tmpsocketv4 == -1)
0740                  {
0741          sprintf( stringBuffer, "===== Error happened in the creation of IPv4 SOCKET in
```

```
AppSIPMobilitySrcAddrSelectionWithDst() =====\n");
0742                            OutputConsoleString(stringBuffer);
0743                            return RV_ERROR_UNKNOWN;
0744                    }
0745
0746                    WSASetLastError(0);
0747   WSAIoctl(tmpsocketv4, SIO_ROUTING_INTERFACE_QUERY, dstIPv4Addr, sizeof(struct sockaddr_in), &addr,
sizeof(struct sockaddr_storage), &ipv4len, NULL, NULL);
0748                    errorcode = WSAGetLastError();
0749                    if(errorcode > 0)
0750                    {
0751            sprintf( stringBuffer, "===== Error happened in WSAIoctl() in AppSIPMobilitySrcAddrSelectionWithDst() with
errorCode %d =====\n\n", errorcode );
0752                            OutputConsoleString(stringBuffer);
0753                            closesocket(tmpsocketv4);
0754                            return RV_ERROR_UNKNOWN;
0755                    }
0756
0757                    //added 200602
0758                    /*
0759                    if( bind(tmpsocketv4, (struct sockaddr *) &addr, sizeof( struct sockaddr_in )) == SOCKET_ERROR )
0760                    {
0761                            fprintf( stderr, "===== Bind: address bind error in AppSIPMobilitySrcAddrSelectionWithDst()" );
0762                            closesocket(tmpsocketv4);
0763                          return RV_ERROR_UNKNOWN;
0764                    }
0765                    */
0766
0767                    closesocket(tmpsocketv4);
0768
0769
0770                    memcpy( srcIPv4Addr, &addr, sizeof(struct sockaddr_storage));
0771
0772            if( getnameinfo((struct sockaddr *)srcIPv4Addr, ipv4len, srcIPv4AddrString, 1024, NULL, 0, NI_NUMERICHOST) !=
0)
0773                    {
0774                            sprintf( stringBuffer, "Error happened in IPv4 getnameinfo...");
0775                            OutputConsoleString(stringBuffer);
0776                    }
0777                    else
0778                    {
0779                            sprintf( stringBuffer, "< Select A Source IPv4 Address... >\n\t%s\n\n", srcIPv4AddrString );
0780                            OutputConsoleString(stringBuffer);
0781                    }
0782
0783            }
0784
0785            if( dstIPv6Addr != NULL && dstIPv6Addr->ss_family == AF_INET6)
0786            {
0787                    SOCKET tmpsocketv6;
0788
0789                    if(dstIPv6Addr->ss_family != AF_INET6)
0790                            return RV_ERROR_UNKNOWN;
0791
0792                    tmpsocketv6 = (int)socket(PF_INET6, SOCK_DGRAM, IPPROTO_UDP);
0793                    if(tmpsocketv6 == -1)
0794                    {
0795            sprintf( stringBuffer, "===== Error happened in the creation of IPv6 SOCKET in
AppSIPMobilitySrcAddrSelectionWithDst()=====\n\n");
0796                            OutputConsoleString(stringBuffer);
0797                            return RV_ERROR_UNKNOWN;
0798                    }
0799
0800                    WSASetLastError(0);
0801                    WSAIoctl(tmpsocketv6, SIO_ROUTING_INTERFACE_QUERY, dstIPv6Addr, sizeof(struct sockaddr_in6),
0802                            &addr, sizeof(struct sockaddr_storage), &ipv6len, NULL, NULL);
0803                    errorcode = WSAGetLastError();
0804                    if(errorcode > 0)
0805                    {
0806            sprintf( stringBuffer, "===== Error happened in WSAIoctl() in AppSIPMobilitySrcAddrSelectionWithDst() with
errorCode %d =====\n\n", errorcode );
0807                            OutputConsoleString(stringBuffer);
0808                            closesocket(tmpsocketv6);
0809                            return RV_ERROR_UNKNOWN;
0810                    }
0811
```

```
0812                    closesocket(tmpsocketv6);
0813
0814                    memcpy( srcIPv6Addr, &addr, sizeof(struct sockaddr_storage));
0815
0816            if( getnameinfo((struct sockaddr *)srcIPv6Addr, ipv6len, srcIPv6AddrString, 1024, NULL, 0, NI_NUMERICHOST) !=
0)
0817                    {
0818            sprintf( stringBuffer, "===== Error happened in IPv6 getnameinfo in AppSIPMobilitySrcAddrSelectionWithDst()
=====\n\n");
0819                            OutputConsoleString(stringBuffer);
0820                    }
0821                    else
0822                    {
0823                            sprintf( stringBuffer, "< Select A Source IPv6 Address... >\n\t%s\n\n", srcIPv6AddrString );
0824                            OutputConsoleString(stringBuffer);
0825                    }
0826
0827            }
0828            return RV_OK;
0829        }
0830
0831        RvStatus AppSIPMobilityDstAddrSelection( const char *remoteAddr, struct sockaddr_storage *dstIPv4Addr,    struct
                                                        sockaddr_storage *dstIPv6Addr,
0832                                                       int *dstIPv4AddrNum, int *dstIPv6AddrNum)
0833        {
0834            WSADATA wsaData;
0835            struct addrinfo hints, *result, *result0;
0836            //struct sockaddr_storage tmpv4, tmpv6;
0837            char addressString[128], portString[8];
0838            int errorCode;
0839            int addrIPv4Num = 0, addrIPv6Num = 0;
0840            bool firstIPv4 = true;
0841            bool firstIPv6 = true;
0842
0843            //The WSAStartup function initiates use of WS2_32.DLL by a process.
0844            if(s_WSAON == RV_FALSE)
0845            {
0846                    if( ( errorCode=WSAStartup( MAKEWORD( 2, 2 ), &wsaData ) )!=0 )
0847                    {
0848            sprintf( stringBuffer, "===== Error happened in WSAStartup() with errorCode %d ====\n\n", errorCode );
0849                            OutputConsoleString(stringBuffer);
0850                            return 1;
0851                    }
0852                    s_WSAON = RV_TRUE;
0853            }
0854
0855            result = NULL;
0856
0857            memset( &hints, 0, sizeof( struct addrinfo ) );
0858            //tmpv4.ss_family = tmpv6.ss_family = 0;
0859
0860            hints.ai_family = PF_UNSPEC;
0861            //The getaddrinfo function provides protocol-independent translation from host name to address.
0862            if( ( errorCode=getaddrinfo( remoteAddr, PORT, &hints, &result0 ) )!=0 )
0863            {
0864            sprintf( stringBuffer, "===== Error happened in getaddrinfo() in AppSIPMobilityDstAddrSelection() with errorCode
%d =====\n\n", errorCode );
0865                    OutputConsoleString(stringBuffer);
0866                    return 1;
0867            }
0868            else
0869            {
0870                    for( result=result0; result!=NULL; )
0871                    {
0872                            switch( result->ai_family )
0873                            {
0874                                    case AF_INET:
0875
0876                                            sprintf( stringBuffer, "< Got A IPv4 Address... >\n" );
0877                                            OutputConsoleString(stringBuffer);
0878
0879                                            if(firstIPv4 == true)
0880                                            {
0881                                                    memcpy( dstIPv4Addr, result->ai_addr, result->ai_addrlen);
0882                                                    firstIPv4 = false;
0883                                            }
```

```
0884
0885        if( ( errorCode=getnameinfo( (struct sockaddr *) dstIPv4Addr, sizeof( struct sockaddr_in ),
0886                                                addressString, 127,
0887                                                portString, 7,
0888                                                NI_NUMERICHOST|NI_NUMERICSERV ) )!=0 )
0889                                        {
0890        sprintf( stringBuffer, "===== Error happened in getnameinfo() AppSIPMobilityDstAddrSelection in IPv4 with
errorCode %d =====\n\n", errorCode );
0891                                                OutputConsoleString(stringBuffer);
0892                                                continue;
0893                                        }
0894                                        sprintf( stringBuffer, "\t%s\n\n", addressString );
0895                                        OutputConsoleString(stringBuffer);
0896                                        addrIPv4Num++;
0897                                        break;
0898
0899                                case AF_INET6:
0900                                        //fprintf( stdout, "====================>>>\n" );
0901                                        sprintf( stringBuffer, "< Got A IPv6 Address... >\n" );
0902                                        OutputConsoleString(stringBuffer);
0903
0904                                        if(firstIPv6 == true)
0905                                        {
0906                                                memcpy( dstIPv6Addr, result->ai_addr, result->ai_addrlen);
0907                                                firstIPv6 = false;
0908                                        }
0909
0910                        if( ( errorCode=getnameinfo( (struct sockaddr *) dstIPv6Addr, sizeof( struct sockaddr_in6 ),
0911                                                addressString, 127,
0912                                                portString, 7,
0913                                                NI_NUMERICHOST|NI_NUMERICSERV ) )!=0 )
0914                                        {
0915        sprintf( stringBuffer, "===== Error happened in getnameinfo() AppSIPMobilityDstAddrSelection IPv6... with
errorCode %d =====\n\n", errorCode );
0916                                                OutputConsoleString(stringBuffer);
0917                                                //return 1;
0918                                                continue;
0919                                        }
0920                                        sprintf( stringBuffer, "\t%s\n\n", addressString );
0921                                        OutputConsoleString(stringBuffer);
0922                                        addrIPv6Num++;
0923                                        break;
0924                                }
0925                                result=result->ai_next;
0926                        }
0927                        freeaddrinfo(result0);
0928                }
0929
0930                (*dstIPv4AddrNum) = addrIPv4Num;
0931                (*dstIPv6AddrNum) = addrIPv6Num;
0932                return 0;
0933        }
0934
0935        RvStatus AppSIPMobilityGetContactHeaderHostName(AppCall *currentCall, RvBool isLocal, char *buf, RvUint32
                                                                *len)
0936        {
0937
0938          char tmpHostAddress[APP_MGR_ADDR_STRING_SIZE];
0939
0940           AppCallGetContactHeaderHostName(currentCall, isLocal, APP_MGR_ADDR_STRING_SIZE, tmpHostAddress,
                                                        len);
0941          //fprintf( stdout, "Remote host: \t%s\n\n", tmpHostAddress );
0942          if(tmpHostAddress[0] == '[')    //IPv6 address
0943          {
0944                  char *pointerS, *pointerE;
0945                  pointerS = strchr(tmpHostAddress, '[');
0946                  pointerE = strrchr(tmpHostAddress, ']');
0947                  if(pointerS != NULL && pointerE != NULL)
0948                  {
0949                          strncpy(buf, (pointerS+1), (pointerE-pointerS-1));
0950                  }
0951          }
0952          else
0953                  strcpy(buf, tmpHostAddress);
0954
0955          return RV_OK;
```

```
0956    }
0957
0958    RvStatus checkAddressAvalibility(sockaddr_storage *srcIPv6Addr, sockaddr_storage *dstIPv6Addr)
0959    {
0960      struct sockaddr_storage raddr, daddr;
0961      SOCKET tmpsocketv6;
0962
0963      tmpsocketv6 = (int)socket(PF_INET6, SOCK_DGRAM, IPPROTO_UDP);
0964      //sprintf( portStr, "%i", 5060 );
0965      memcpy( &raddr, srcIPv6Addr, sizeof( struct sockaddr_storage ) );
0966      memcpy( &daddr, dstIPv6Addr, sizeof( struct sockaddr_storage ) );
0967      //raddr.sin6_family = AF_INET6;
0968      //raddr.sin6_port = htons((short)0);
0969      if( bind( tmpsocketv6, ( struct sockaddr *)&raddr, sizeof( raddr ) ) < 0 )
0970      {
0971              sprintf( stringBuffer, "Error happened in bind...\n");
0972              OutputConsoleString(stringBuffer);
0973      }
0974      else
0975      {
0976              sprintf( stringBuffer, "BIND Address OK...\n");
0977              OutputConsoleString(stringBuffer);
0978      }
0979
0980      if( connect(tmpsocketv6, (struct sockaddr *)&daddr, sizeof(daddr) ) < 0)
0981      {
0982              sprintf( stringBuffer, "Error happened in connect...\n");
0983              OutputConsoleString(stringBuffer);
0984      }
0985      else
0986      {
0987              sprintf( stringBuffer, "connect OK...\n");
0988              OutputConsoleString(stringBuffer);
0989      }
0990
0991      closesocket(tmpsocketv6);
0992      return RV_OK;
0993
0994    }
0995
0996    RvStatus AppSIPMobilityAddNewCall()
0997    {
0998      //void*               pNewObj             = NULL;
0999      AppCall *    tmpCall;
1000      RvStatus rv;
1001
1002      AppCallSetGlobalBody();
1003      rv = AppCallCreate(&tmpCall);
1004      AppCallSetPartyHeader(tmpCall, RV_FALSE, "From: <sip:944021306@sip.ipv6.club.tw>");
1005      AppCallSetPartyHeader(tmpCall, RV_TRUE, "To: <sip:944021307@sip.ipv6.club.tw>");
1006      AppCallSetContactHeader(tmpCall, RV_TRUE, "sip:944021306@sip.ipv6.club.tw");
1007      AppCallSetContactHeader(tmpCall, RV_FALSE, "sip:pc2.ipv6.club.tw");
1008
1009      AppCallConnect(tmpCall);
1010      return rv;
1011    }
1012
1013    RvStatus AppSIPMobilityListAddrQuery(char *localIPv4Addr, char *localIPv6Addr, bool *ipv4Exist, bool *ipv6Exist)
1014    {
1015      int                                 i;
1016      int                                 errorCode;
1017      char                        addressString[128], portString[8];
1018      SOCKET                              sockQuery;
1019      unsigned long               lTmp;
1020      LPSOCKET_ADDRESS_LIST       salNetCards;
1021
1022      sockQuery = WSASocket( AF_INET, SOCK_RAW, IPPROTO_IP, 0, 0, 0 );
1023
1024      if( sockQuery != INVALID_SOCKET )
1025      {
1026              WSAIoctl( sockQuery, SIO_ADDRESS_LIST_QUERY, NULL, 0, NULL, 0, &lTmp, NULL, NULL );
1027              salNetCards = (SOCKET_ADDRESS_LIST*)malloc( lTmp );
1028
1029              if( WSAIoctl( sockQuery, SIO_ADDRESS_LIST_QUERY, NULL, 0, salNetCards, lTmp, &lTmp, NULL,
                NULL ) != SOCKET_ERROR )
1030              {
```

```
1031                    for( i = 0; i < salNetCards->iAddressCount; i++ )
1032                    {
1033                            if( ( errorCode=getnameinfo( salNetCards->Address[i].lpSockaddr, sizeof( struct sockaddr_in ),
1034                                    addressString, 127,
1035                                    portString, 7,
1036                                    NI_NUMERICHOST|NI_NUMERICSERV ) )!=0 )
1037                            {
1038                                    sprintf( stringBuffer, "===== Error happened in getnameinfo()
                                        AppSIPMobilityDstAddrSelection in IPv4 with errorCode %d =====\n\n", errorCode );
1039                                    OutputConsoleString(stringBuffer);
1040                                    continue;
1041                            }
1042                            sprintf( stringBuffer, "\t%s\n\n", addressString );
1043                            OutputConsoleString(stringBuffer);
1044                            if(strcmp(addressString, localIPv4Addr) == 0)
1045                            {
1046                                    fprintf(stderr, "IPv4 Address Exist\n");
1047                                    (*ipv4Exist) = true;
1048
1049                            }
1050
1051                    }
1052            }
1053            free( salNetCards );
1054             closesocket( sockQuery );
1055    }
1056
1057    sockQuery = WSASocket( AF_INET6, SOCK_RAW, IPPROTO_IP, 0, 0, 0 );
1058
1059    if( sockQuery != INVALID_SOCKET )
1060    {
1061            WSAIoctl( sockQuery, SIO_ADDRESS_LIST_QUERY, NULL, 0, NULL, 0, &lTmp, NULL, NULL );
1062            salNetCards = (SOCKET_ADDRESS_LIST*)malloc( lTmp );
1063
1064            if( WSAIoctl( sockQuery, SIO_ADDRESS_LIST_QUERY, NULL, 0, salNetCards, lTmp, &lTmp, NULL,
                NULL ) != SOCKET_ERROR )
1065            {
1066                    for( i = 0; i < salNetCards->iAddressCount; i++ )
1067                    {
1068                             if( ( errorCode=getnameinfo( salNetCards->Address[i].lpSockaddr, sizeof( struct
                                        sockaddr_in6 ),
1069                                    addressString, 127,
1070                                    portString, 7,
1071                                    NI_NUMERICHOST|NI_NUMERICSERV ) )!=0 )
1072                            {
1073                                    sprintf( stringBuffer, "===== Error happened in getnameinfo()
                                        AppSIPMobilityDstAddrSelection IPv6... with errorCode %d =====\n\n", errorCode );
1074                                    OutputConsoleString(stringBuffer);
1075                                    continue;
1076                            }
1077                            sprintf( stringBuffer, "\t%s\n\n", addressString );
1078                            OutputConsoleString(stringBuffer);
1079                            if(strcmp(addressString, localIPv6Addr) == 0)
1080                            {
1081                                    fprintf(stderr, "IPv6 Address Exist\n");
1082                                    (*ipv6Exist) = true;
1083                            }
1084
1085                    }
1086            }
1087
1088            free( salNetCards );
1089             closesocket( sockQuery );
1090    }
1091
1092    return RV_OK;
1093
1094  }
1095
1096  /*
1097
1098  RvStatus AppSIPMobilityGetNameInfo(struct sockaddr_storage *addr, char *buf)
1099  {
1100
1101    if( getnameinfo((struct sockaddr *)addr, sizeof( struct sockaddr ), buf, 1024, NULL, 0, NI_NUMERICHOST) != 0)
1102    {
```

```
1103                    fprintf( stderr, "Error happened in IPv6 getnameinfo...");
1104                    return RV_ERROR;
1105            }
1106            return RV_OK;
1107        }
1108        */
1109        //==================Waste=====================
1110
1111        /*
1112            struct sockaddr_in    raddr;
1113            //sprintf( portStr, "%i", 5060 );
1114            memset( ( void * )&raddr, 0, sizeof( struct sockaddr_in ) );
1115            raddr.sin_family = AF_INET;
1116            raddr.sin_addr.s_addr = htons( INADDR_ANY );
1117            raddr.sin_port = htons((short)0);
1118            if( bind( tmpsocketv4, ( struct sockaddr * )&raddr, sizeof( raddr ) )<0 )
1119                    fprintf( stderr, "Error happened in bind...");
1120        */
1121
1122        //#endif
```