USING THE SESSION INITIATION PROTOCOL AS A NETWORKING PROTOCOL FOR HOME APPLIANCES

by

Josué A. Díaz Torres

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGUEZ CAMPUS
2005

Approved by:

_____	_____
Isidoro Couvertier Reyes, Ph. D. President, Graduate Committee	Date
_____	_____
Manuel Rodríguez Martínez, Ph. D. Member, Graduate Committee	Date
_____	_____
Jorge Cruz Emeric, Ph. D. Member, Graduate Committee	Date
_____	_____
Rafael A. Olivencia Martínez, MBA Graduate Studies Representative	Date
_____	_____
Isidoro Couvertier Reyes, Ph. D. Chairperson of the Department	Date

Abstract

Home appliances have evolved from being single task devices to integrating several tasks in a particular device. The next step for home appliances is to transform into networked appliances and accelerate the development of home automation. This research had the objective of adding a new method, called DO, to the Session Initiation Protocol (SIP) in order to control these networked appliances. Considering that SIP methods are self explanatory, the name of this new method does not deviate from this convention. DO is a verb that means to carry out or perform an action and this is exactly what the method represents. Once the method was added to a SIP stack, it was tested by sending a DO request to a SIP User Agent representing a networked appliance. The results for these tests were the same as the expected; the SIP User Agent executed the action that was requested from it in the DO message, thus fulfilling the research objective.

Resumen

Los enseres del hogar han evolucionado de aparatos que realizaban una sola tarea, a integrar diversas tareas en un solo aparato. El próximo paso para los enseres del hogar es su transformación a enseres con conexión a la red y acelerar el desarrollo de la automatización del hogar. El objetivo de esta investigación era añadir un nuevo método, llamado "DO", al "Session Initiation Protocol" (SIP) para controlar estos enseres con conexión a la red. Debido a que los nombres de los métodos de SIP se explican por sí mismos, el nombre de este nuevo método no se aleja de esta convención. "DO", en el idioma inglés, es un verbo que significa realizar una acción, que es exactamente lo que el método representa. Una vez añadido el método a un "stack" de SIP, se probó enviando un mensaje "DO" a un "user agent" de SIP que representó un enser con conexión a la red. Los resultados de las pruebas fueron idénticos a los resultados esperados, el "user agent" ejecutó la acción que se le indicó en el mensaje "DO" cumpliendo de esta forma con el objetivo de la investigación.

Acknowledgements

I would like to thank God for giving me the strength to complete this milestone, and especially my parents for their constant love and support, this is for you.

Thanks to my graduate committee chairman, Dr. Isidoro Couvertier, for his guidance, advice, and support. I would also like to thank the rest of my graduate committee, Dr. Manuel Rodríguez and Dr. Jorge Cruz Emeric, for their patience and understanding.

Special thanks to Commoca Inc., Dr. José Meléndez, Dr. José Luis Cruz, and the technical staff for believing in me and letting me finish my research project using their product.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

Home appliances were originally designed to perform specific tasks and in the early years of their existence they did. The refrigerator, television, washer, dryer, among others, all they did was satisfy particular needs. As science and technology has evolved, so have these appliances especially with the introduction of microprocessors and microcontrollers. What started as purely mechanical devices, are now integrating memory and control chipsets. For example, washers evolved from having a single washing cycle to having three or four cycles, each one of them with a selectable washing time, among other options. Manufacturers have added more functionality to home appliances by integrating features provided by different devices into one device. Televisions have incorporated the functionality of an alarm clock, stoves display the time, and phones have integrated answering machines.

Personal computers, on the other hand, were designed to satisfy a general purpose, they are not usually optimized to perform a particular task. The type of work performed by home appliances cannot be done by a personal computer, but being able to make them work together effectively is the key to the next generation of home appliances.

The next step in the development of these appliances is the ability to control them remotely via the Internet and inter-device communication. In order to accomplish these tasks, home appliances need an embedded processor and a network connection

[Moyer2000b]. Devices with these characteristics will be referred to as networked appliances for the rest of this work.

Networked appliances and their ability to work together give way to home automation which has encountered a road block because of the lack of effective communication between different types of appliances. Although there are several technologies available for home automation, there is a lack of support for these networked appliances to work together. Some companies have been developing technologies for home automation and use them effectively in smart homes.

There is an ongoing effort to develop smart homes, for example, Microsoft has a smart home prototype on its campus in Redmond, Washington. Some of its features include a welcoming screen that lets visitors choose between ringing the bell or leave a message that gets broadcasted throughout the home messaging system besides sending an e-mail to the household. Access to the home is controlled by an iris scanner and a smart card reader. The appliances are controlled from a remote control that lets the user customize his or her settings. For example, activating the "Movie" setting can be customized to dim the lights, lower the window shades and set the audio volume to the preferred level. Even though the appliances can work together, they operate independently of one another. The Microsoft Smart Home uses technologies in development and the electronic devices use the Universal Plug and Play protocol (UPnP) and the eXtensible Markup Language (XML).

Another global company that has spent some effort on the development of smart homes is International Business Machines (IBM). Along with a housing developer firm

and some other partners, IBM constructed 170 houses at The Village in Tinker Creek in Roanoke, Virginia. Each home has an integrated system with "smart" technology for remote management of heating, air conditioning, and security systems [Wrolstad2003]. The Web and mobile access is offered using IBM's WebSphere platform.

Both examples mentioned required the construction of new homes and additional infrastructure. The purpose and contribution of this research is to implement an extension method to the Session Initiation Protocol in order to control networked appliances using existing equipment, existing infrastructure and a lightweight protocol. This method called DO, meaning to perform an action, was proposed by Tsang [Tsang2000a] but no implementation has been documented to this date. The research not only will highly contribute to the home automation industry, but also on the development of smart homes and sun light powered homes. Using networked appliances in solar homes provides a high level of power management. The ability of checking a device status lets the home controller decide whether this device should be on or not or even decrease the power consumption of this device.

Chapter two describes the Session Initiation Protocol and previous work that has been documented in the area of this research. The details of design and implementation are discussed on chapter three followed by the documentation of results gathered in the study on chapter four of this work. Finally conclusions and future work are documented on chapter five.

Chapter 2. Literature Review

The idea of home appliances communicating through a network has been in study for several years. Telcordia Technologies was one of the first to propose the idea of networked appliances and studied the market for this type of devices. Their first case study considered an Internet alarm clock. This study was based on studying the implications that networked appliances had on network requirements and design.

2.1 The Internet alarm clock

Moyer and Marples [Moyer2000a] identified two features that could be added to a traditional alarm clock. The first improvement consisted on configuring rules on the alarm clock which the device would use to program itself given a target time. For example, if the user wants to be at work at a given time, the alarm clock uses the time it takes the user to get dressed, eat breakfast, and drive to work to program itself. The second area of improvement identified was adding network connectivity to the alarm clock so that, besides considering the rules configured by the user, it would use external factors to calculate the appropriate wake up time.

The alarm clock was constructed using an LCD display and an internet interface board, which had the clock driver and clock controller. It was also connected to a web server which interpreted the service rules and contained user profile data. The user entered the time at which he wanted to arrive at work the next day, the system then computed at what time it needed to set the alarm depending on distance to travel, traffic

and weather conditions, and the time it takes the user to get ready, this last data is entered by the user.

It was found that the alarm clock offered useful functionality; the biggest problem was the changing format of the web sites used as information sources [Moyer2000a]. Some issues did rise from this experiment, such as the communication across the boundary of the home network via a Network Address Translator. The transfer of data outside the home domain causes problems with device addressability, service naming and location, and security. These problems led to the development of some requirements to provide wide-area access control and interworking support to networked appliances.

2.2 Requirements for network appliances

There are several technologies that provide home automation, to some extent, these home networking technologies include, among others, X.10, UPnP, and HAVi. However, these technologies do not provide control of the appliances from the Internet; neither do they provide a way of interworking with one another. In order to solve these problems a set of requirements had to be established. The requirements mentioned in [Tsang2000a] are documented in the following sub-sections.

2.2.1 General Requirements

General requirements for networked appliances include wide area accessibility; the device must be accessible from outside the home Local Area Network (LAN). Regarding local communication, networked appliances must interwork with different

home networking technologies in a transparent manner. The use of additional controllers is required to connect the networked appliances to the Local Area Network. These controllers, as shown in Figure 1, can provide interworking between the home networking technologies and the Internet Protocol [Tsang2000a]. It is not necessary to have one controller for each appliance; a single controller can manage a network of devices. For example, the TV, the VCR, the DVD player and the stereo can communicate between themselves using the HAVi protocol. This HAVi network can be connected to the LAN using a HAVi - LAN appliance controller which makes it possible to communicate with other appliances in the network. The controller should translate the action received in a LAN message to the corresponding HAVi message and vice versa.

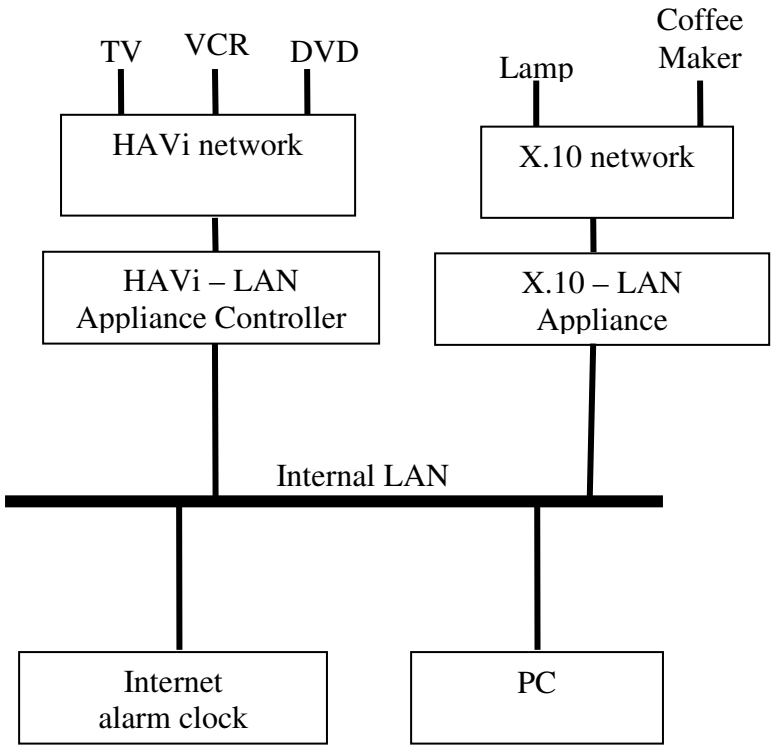

Figure 1. Interwork of different technologies in the home LAN.

Networked appliances should also be able to auto-configure themselves with minimal or no user interaction, although manual configuration should be allowed. If the user wishes to configure the appliance, the networked appliance controller should provide a configuration webpage that the user can access from his or her personal computer.

2.2.2 Communication Protocol Requirements

The communication protocol has to provide flexible data capability, meaning that the protocol should allow for different size of data. Reliability against communication errors and communication breakdown must also be provided as well as event notification.

Messages for networked appliances control are expected to be short as well as sessionless; because of this the User Datagram Protocol is the ideal transport protocol. These messages should contain appliance characteristics regarding response time and capabilities. In summary, the communication protocol is required to support control messages, queries, asynchronous events, discovery messages, and support media streaming.

2.2.3 Security Requirements

Connecting home appliances to the outside world can become a problem if the necessary security considerations are not taken into account. If the user does not have a strong security policy, it is possible for people outside the home environment to control the networked appliances.

Authentication, authorization, privacy, and replay protection are required in all appliance communications [Tsang2000a]. Authorization checks may be done per registration, per message or from time to time. If the user does not want a certain appliance to be accessed from outside the home LAN, it should be possible to isolate networked appliances from external networks. Even if the networked appliance is isolated from the Internet, it must be able to communicate with networked appliances inside the LAN and perform its function correctly.

Once these requirements were established it was time to find a suitable networking protocol. The protocol proposed by Telcordia Technologies, that met the requirements identified above, was the Session Initiation Protocol (SIP), but in order to understand how SIP can help to communicate and control home appliances it is necessary to understand the protocol itself.

2.3 Session Initiation Protocol

The Internet Engineering Task Force (IETF) defines the Session Initiation Protocol, or SIP, as "an application-layer control that can establish, modify, and terminate multimedia sessions (conferences) such as Internet telephony calls" [Rosenberg2002]. SIP works with existing protocols by enabling endpoints, called user agents, to discover other user agents and agree in the type of session they would like to share. The type of session is specified using the Session Description Protocol (SDP). Sinnreich and Johnson [Sinnreich2001] state that SDP is not a true protocol; it is a text-based description language.

A user agent is the entity that interacts with the user, it can run as a computer program or it can be implemented on a single device. These entities can be classified as SIP clients or SIP servers. User agents communicate with each other via text-based messages. A brief description of SIP functionality can be found in Appendix A.

2.3.1 SIP messages

SIP is a text-based protocol, as it was mentioned above. Messages are either requests or responses. A SIP transaction is consists of a request, one or more provisional responses, and a final response. SIP requests contain a field, called a method, which describes its purpose [Camarillo2002] and it is part of the request line. The IETF defines six types of requests

- **INVITE**: Invite other users to participate in a session. It contains the description of the session.
- **ACK**: Used to acknowledge the reception of a final response to an INVITE, this provides for a three-way handshake between user agents.
- **CANCEL**: This request cancels pending transactions. If the user agent receiving this request has not sent a final response to a request, it stops processing the request.
- **BYE**: Request used to terminate a session between two parties. If the session has more than two parties, the user agent that sends these requests abandons the session.

- **REGISTER**: Used to indicate a SIP server about the current location of the user agent.
- **OPTIONS**: A client sends this request to query a server about which methods and which session description protocols it supports, as well as other capabilities.

Once a user agent receives a request, it issues one or more responses. SIP responses contain a numeric code and a reason phrase that are part of the status line in the message. A table with SIP responses classes is shown in Table 1 [Camarillo2002].

Table 1. SIP response range and classes

Response Numeric Range	Response Class
100-199	Informational
200-299	Success
300-399	Redirection
400-499	Client Error
500-599	Server error
600-699	Global failure

SIP requests and responses contain headers that provide information about the request or response and about the message body [Camarillo2002]. Some examples of headers relevant to this research are

- **From**: Contains the request sender and its SIP Universal Resource Locator (URL) or SIP address. An example of this type of header is shown below:

  From: Josue Diaz <sip:Josue.Diaz@uprm.edu>

- **To**: Indicates the recipient of the request, it contains the address of the destination party.

  To: Joe Johnson <sip: Joe.Johnson@company.com>

- **Cseq**: Command Sequence header. It consists of a numerical part and a method name. The numerical part orders requests within the same session.

  Cseq 25 INVITE

- **Via**: Used as a routing mechanism. Messages can contain several Via headers, one for each proxy server that handled the request. It helps to route the responses back to the request originator.

  Via: SIP/2.0/UDP 192.168.10.1

- **Content-Type**: Provides information about the message contained in the body of the SIP message.

  Content-type: application/sdp

- **Content-Length**: Contains the length in bytes of the SIP message body.

  Content-Length: 150

The body of the SIP messages as well as how it is handled depends on the application. Most bodies contain the description of the session using SDP, but they can also contain plain text. This type of flexibility makes SIP an ideal protocol for Internet telephony, instant messaging and, in the case of this work, controlling networked appliances.

2.3.2 SIP extension to control networked appliances

The six methods mentioned earlier are the foundation of SIP, but none of them has the capability of carrying control messages to a networked appliance. A new excitation method, called DO was proposed in [Tsang2000b]. Considering that the names of SIP methods are self explanatory, as shown in the methods explained earlier, the name of this new method does not deviate from this convention. DO is a verb meaning to carry out or perform an action and this is exactly what the method represents. This method would enable request to be sent without setting up a new session between user agents, but this does not mean that it cannot be used within an existing session. In this case, the user agents involved would be the appliance controllers, computer programs and a SIP server.

DO requests contain a message body which carries the action to be performed by the networked appliance. An example of a DO request is shown below.

```
DO sip: bedroomLamp@josuehome.net SIP/2.0
From: sip: JosuePC@josuehome.net
To: sip: bedroomLamp@josuehome.net
Via: SIP/2.0/UDP josuehome.net
Cseq: 25 DO
Content-Type: application/text
Content-Length: 8

        Turn on.
```

In this example the user agent in the personal computer sends a DO request to the bedroom lamp appliance controller. The control message indicates the lamp to turn on.

A user agent receiving a DO request responds with a 200 OK response message, meaning that it received the request successfully. The body of the response message, as stated in [Tsang2000b], may contain additional information regarding the status of the device. A response message to the DO request in the example above should be as follows.

```
SIP/2.0 200 OK JosuePC@josuehome.net
From: sip: JosuePC@josuehome.net
To: sip: bedroomLamp@josuehome.net
Via: SIP/2.0/UDP josuehome.net
Cseq: 25 DO
Content-Length: 0
```

The appliance controller responds with a 200 OK and copies the headers from the request message. The header of Content-Length: 0 indicates that the responds contains no body.

2.4 SIP as a home appliances networking protocol

The decision of using SIP as a protocol to communicate and control home appliances was based on the functionality provided by SIP compared to the requirements stated in section 2.2. Moyer in [Moyer2001] discusses some characteristics of SIP that make it an ideal protocol for this task.

- **Interoperability**. Enables communication with devices in the local area network independent of the type of local device communication protocol being used.
- **Security**. Provides authentication and encryption. An authorization check can be performed before executing the action in the SIP request. It also has means of encrypting the payload to provide privacy from user agent to user agent.

- **Scalability**. Very scalable protocol. It works well in both Local Area Network and Wide Area Network environments. One of the most important characteristics of SIP is that it is independent of the transport protocol; it works with the Transport Control Protocol (TCP) as well as with the User Datagram Protocol (UDP).
- **Mobility**. SIP supports mobility concept, meaning that a user agent can move from one environment to another and still be found. This is possible because of the REGISTER method defined in section 2.3 which informs a SIP server of the user agent new location.
- **Extensibility**. It allows new methods, new message types, new type or address forms, and it supports the four types of communication modes identified for Networked Appliances: control, query, event notification, and multimedia session.
- **Service Convergence**. SIP uses existing infrastructure which makes it easier to administer and maintain. The ability to manage only one infrastructure for a variety of services, including controlling networked appliances makes SIP a very attractive protocol.

Chapter 3. Methodology

The design and implementation of this work was done using a SIP stack provided by Commoca, Inc. Commoca's VoIP (Voice over IP) phone platform served as a test bed for this research, none of the phone features were used, only its SIP stack. To avoid confusion, from this point on the phone will be called the SIP User Agent, since it is the interface between the user and the SIP stack and it will represent a networked appliance. Both the SIP User Agent and the SIP stack are programmed in C and compiled using the GNU C/C++ compiler. The following sections describe the process of adding the DO method to the stack.

3.1. SIP stack architecture

The SIP User Agent stack supported the six methods mentioned in section 2.3.1, REGISTER, OPTIONS, INVITE, ACK, CANCEL, and BYE. In order to successfully add a method to this stack, it was necessary to understand the stack architecture.

The SIP stack consists of four layers, application layer, control layer, transaction layer, and transport layer. The application layer is where the user application resides; any code related to the user agent functionality has to be added at this level. An application needs to call two methods before attempting to access the stack. These methods are the stack initialization method and the set callback functions method. The stack initialization method initializes the SIP stack configuration with the default parameters, constructs a stack with this configuration and it assigns the application a stack handler as well as a transaction manager handler. These handlers identify the application that is accessing the

stack or asking for a transaction at any moment. A flow diagram of this process is shown in Figure 2. The set callback functions method initializes the functions that will be called when a transaction message has been received or when there is a state change.

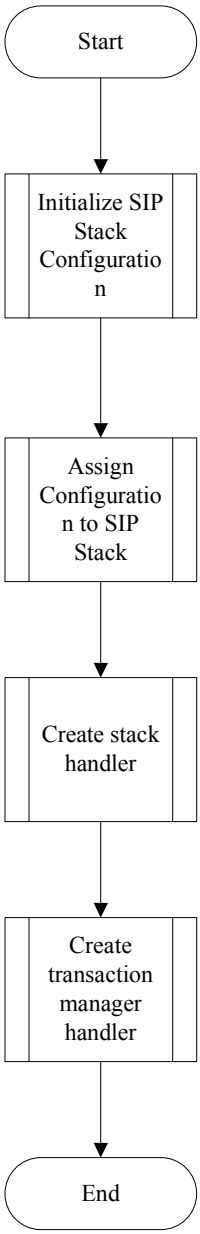

Figure 2. Initialization process flow diagram.

Stack control methods are inside the control layer; this layer is also called the stack manager. The stack manager contains the stack initialization routine and is in charge of distributing the handlers for the lower layers.

The transaction layer is in charge of handling the SIP stack methods. If the User Agent wants to make a call to another User Agent, this layer processes the INVITE request and sends back the response messages. This layer communicates directly with the transport layer which is the responsible for sending and receiving the UDP messages.

3.2 Adding the DO method to the SIP stack

Since the objective of this research was extending the SIP stack by adding a DO method to control networked appliances, the major part of the code was added in the transaction layer although some code was added to the application and control layers. Pseudo code of this work can be found in Appendix B.

The first step was adding the DO method to the stack manager (SIP_METHOD_DO) and to the transaction layer (SIP_TRANSACTION_METHOD_DO) so that it would be recognized by the SIP stack as a SIP method. The new method also needs to have transaction states associated with it; these states were added to the SIP transaction states list. The names of these new states are SIP_TRANSC_STATE_CLIENT_DO_REQUEST_RCVD and SIP_TRANSC_STATE_CLIENT_DO_REQUEST_SENT. Once the stack is aware of the existence of the DO method the handlers for the DO transaction were added.

When the SIP stack receives a transaction request, it handles the request depending on the request method. When a transaction request is received, the stack calls a request handling routine that checks for the request method. The DO method was added as one of the options in this routine. If the transaction request is a DO request, the check routine calls a function called HandleDoInInitialState. This function is responsible for calling the transaction message received callback function and changing the transaction state to SIP_TRANSC_STATE_CLIENT_DO_REQUEST_RCVD.

The code added in the application layer was basically three callback functions, AppTranscStateChangedEvHandler, AppTranscMsgReceivedEvHandler, and AppTranscMsgToSend. These functions are called from the transaction level as it was mentioned above. As shown in Figure 3, callback AppTranscStateChangedEvHandler sends a 200 OK response message indicating that the request was received, if the state was set to SIP_TRANSC_STATE_CLIENT_DO_REQUEST_RCVD. The application that sends the DO request must resend the transaction request if it does not receive a 200 OK response.

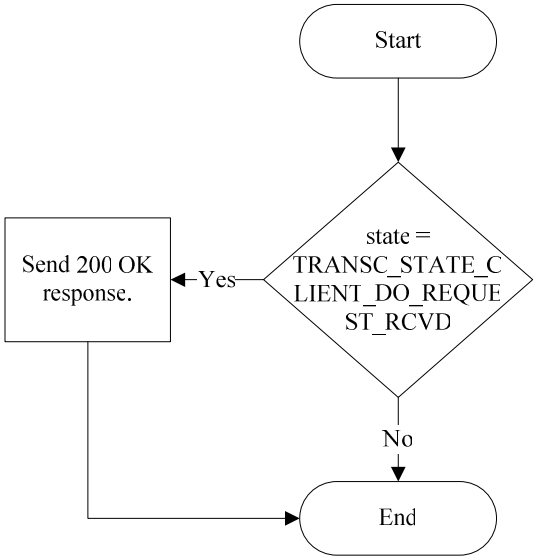

Figure 3. Transaction state changed callback flow diagram.

The second callback, AppTranscMsgReceivedEvHandler, checks if the request method is a DO request, if it is it parses the DO SIP message. Since this function is only interested in the body of the SIP message, once the message is parsed, it extracts the message body that contains the action to be done by the SIP User Agent. It compares the action to the list of actions that the User Agent can perform. If the action is valid, the function calls the application controller to perform the requested action. If the action in the SIP message body is not a valid action, the callback simply returns. A flow diagram of this callback is shown in Figure 4.

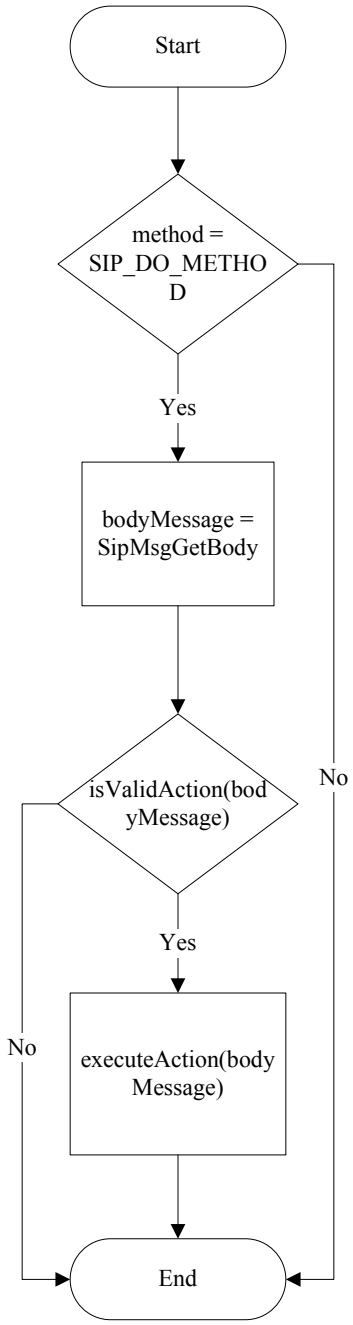

Figure 4. Transaction message received callback flow diagram.

In order to send a DO request message the application must first set the parameters for the DO message. This is done by calling function setDoRequestMessage

which receives the destination user, destination address, sender address and the action to be performed. Once the values have been set, the application calls function AppSendDoMsg. This function is in charge of getting a transaction manager handler from the control layer. Using this handler, the application asks the transaction layer to create the transaction message using the parameters set before and to send the transaction request via the transport layer.

When the transaction layer is constructing the request message it calls the AppTranscMsgToSend callback. The purpose of this callback is to set the Content-Type header and the request message body. For a DO request, the callback function will set the Content-Type header to **application/text** meaning that the body contains a text message to be handled by an application. The callback will also fill the message body with the action set in function setDoRequestMessage. The flow diagram for this callback is shown in Figure 5. Once the message is constructed and verified by the transaction layer, it is sent to the transport layer which opens a socket and sends the UDP packets containing the SIP DO request.

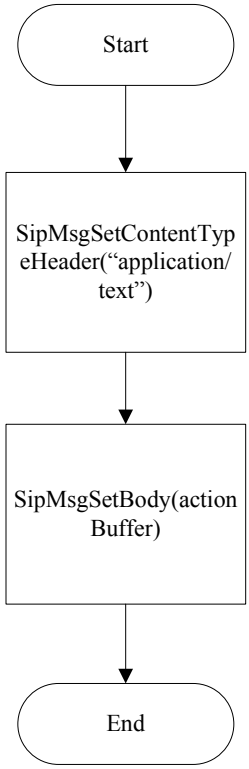

Figure 5. Message to send callback flow diagram.

3.3 Testing the DO method

Testing of the DO method added to the SIP stack was done using two SIP User Agents in a LAN; the SIP User Agents represented networked appliances. The SIP User Agents were connected to a LAN and communicated peer-to-peer, the term peer-to-peer in SIP means that the user agents involved in a transaction send messages directly to one another without the intervention of a SIP proxy server.

The SIP User Agents used for testing provided a text-based interface in which the symbols of the image could be executed from the command prompt via telnet or via the Agent's serial port. A personal computer (PC) was used to communicate to the SIP User

Agents using the program Tera Term Pro. A diagram of the testing environment is depicted in Figure 6. The testing was done with both, telnet and serial port communication.

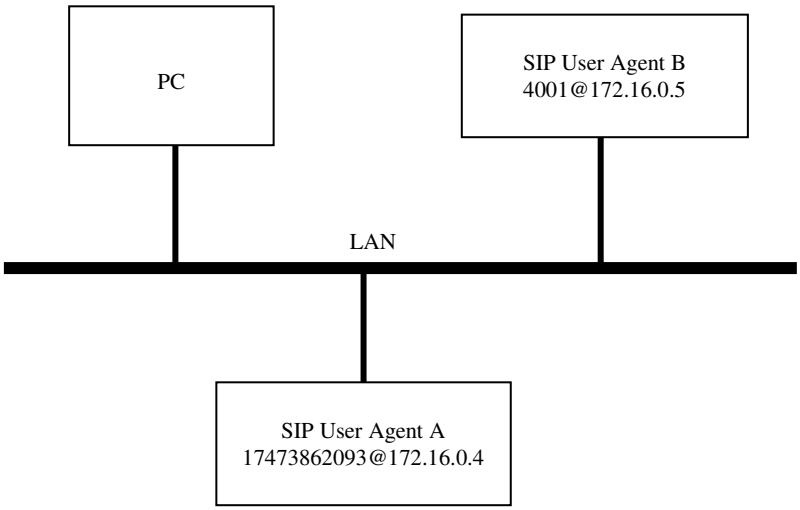

Figure 6. Testing environment diagram.

As described in section 3.3, in order to send a DO request message, the SIP User Agent needs to set the parameters using function setDoRequestMessage and then send the request using function AppSendDoMsg. Both of these functions are documented in Appendix B. Function setDoRequestMessage has the following parameters: destination user, destination address, source address and the action to be performed. Function AppSendDoMsg has no parameters.

For testing purposes the functions mentioned above where called from the text-based interface. Once the communication to SIP User Agent A was established from the PC the setDoRequestMessage was called with parameters: destination user = 4001, destination address = 172.16.0.5, source address = 172.16.0.4, action = “Start Ringer”.

After successfully setting the DO request parameters, function AppSendDoMsg was called from the command prompt to send the DO request to SIP User Agent B.

The SIP messages were captured using the Ethereal Network Analyzer and the results of this test are documented in the next chapter.

Chapter 4. Results and Discussion

The DO method implementation worked as expected. SIP User Agent A sends a DO request to SIP User Agent B, SIP User Agent B responds with a 200 OK and executes the requested action. Figure 7 shows the expected behavior described above

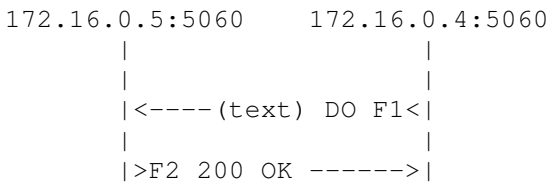

Figure 7. DO method test expected behavior.

The analysis of the Ethereal capture in Figure 8 shows that the DO message was constructed as described in section 2.3.2. The destination is SIP User Agent B with user 4001 and a destination address of 172.16.0.5. SIP User Agent A which is requesting the action has user 17473862093 and address 172.16.0.4. The requested action contained in the SIP message body for the test is the string “Start Ringer”. If the networked appliance were a bedroom lamp, the SIP message body could have been “Turn on”.

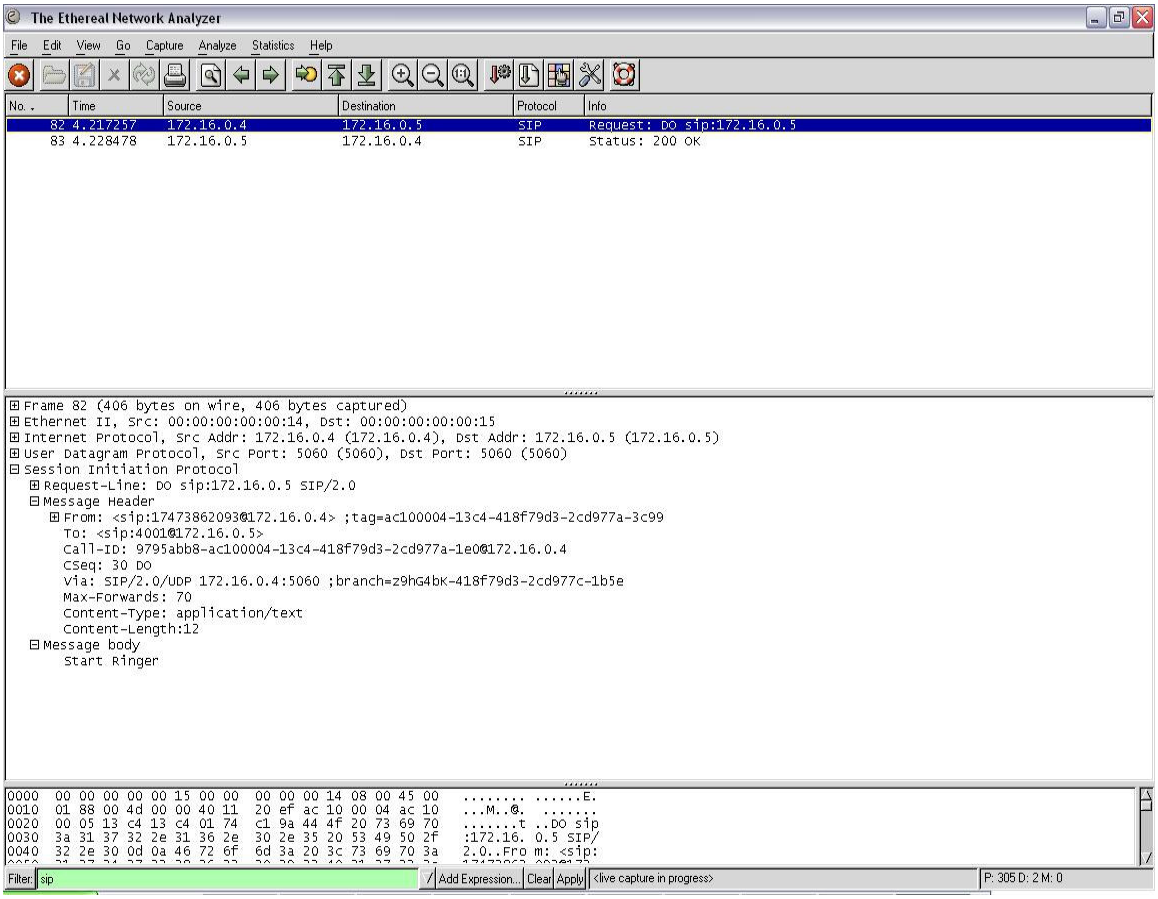

Figure 8. Ethereal DO request capture.

Once SIP User Agent B, with user 4001, representing a networked appliance, receives the DO request it responds with a 200 OK as shown in Figure 9. After it extracts the action to be performed and verifies that it is valid, it executes the requested action received from SIP User Agent A. In this case SIP User Agent B starts to ring after it receives the DO message which indicates that the test was successful. SIP User Agent B received the request, checked that the action requested was valid and proceeded to execute it, which was the expected behavior mentioned earlier.

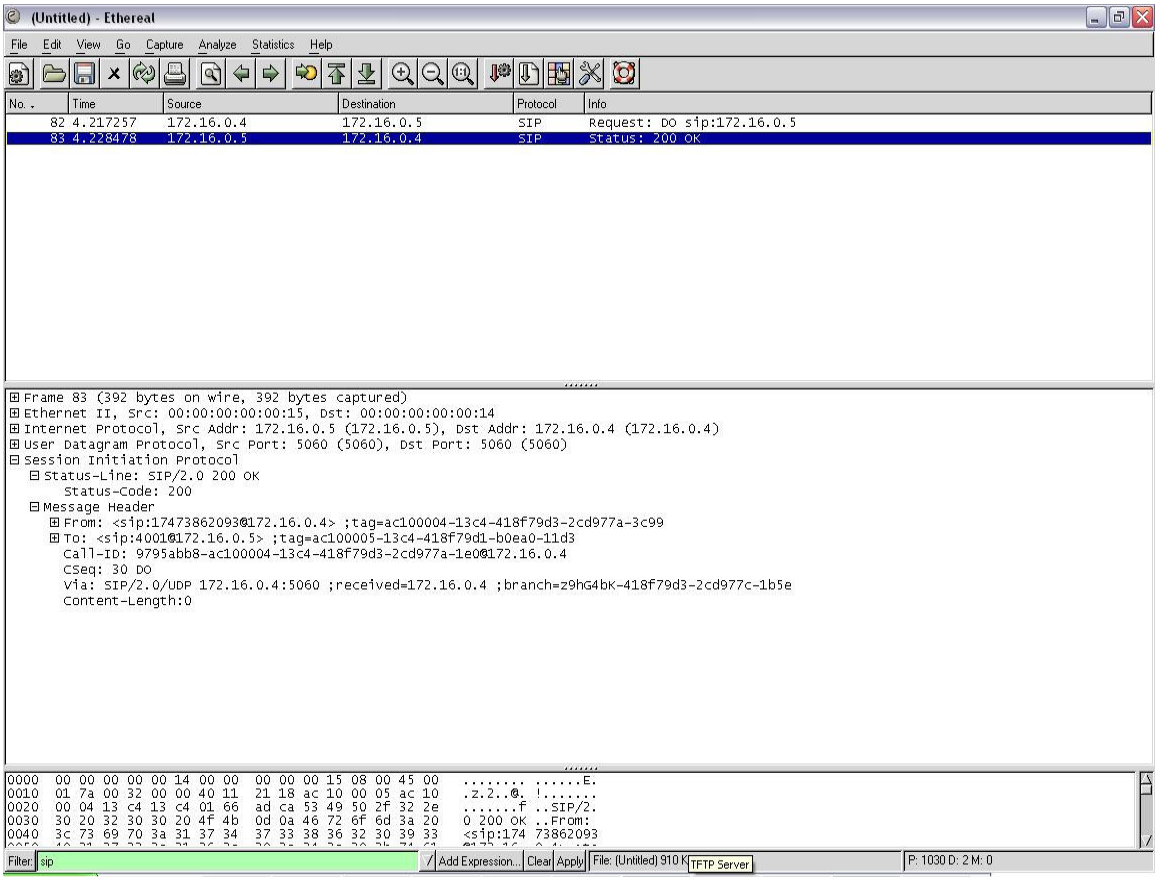

Figure 9. Ethereal 200 OK response capture.

The process of design and development using the SIP stack provided by Commoca, Inc. was slow at the beginning because of the learning curve. The process of getting acquainted with the stack, its architecture and object oriented programming style written in C made it difficult to understand. Once this long process was completed, the implementation of the DO method hit a roadblock when the message was being sent by SIP User Agent A but it received no answer from SIP User Agent B. After debugging the problem, the root of the issue was found to be that the callback that handles the messages received from transactions was not implemented correctly in the application level, meaning that the message was being received but not handled.

Chapter 5. Conclusions and future work

This work has demonstrated the capabilities of the Session Initiation Protocol and how, by the addition of a single method, it is able to control networked appliances. The ability of SIP being a protocol independent of the transport protocol underneath it, as well as the benefit that it uses existing infrastructure, makes it an ideal protocol to interwork devices that use different networking protocols. This is critical for home appliances since there are existing home automation protocols but none of them can interact with one another.

The study suggests that the only additional hardware needed is an appliance controller that works as a SIP user agent to connect an appliance or a network of appliances of similar technology to the home LAN. This controller, besides being SIP compliant, should have some kind of routing capability and it needs to translate the action received in a SIP message to the given technology. For example, if the controller is a X.10 – SIP appliance controller, it must be able to translate the action to a X.10 command for the specified appliance. The number of appliances that can be connected to a single controller will depend on the controller's capabilities and specifications, which are not covered in this work.

The scalability for big homes with a large number of appliances depends on the number of appliances that the controllers can manage. Instead of having a single controller for each appliance, dividing the appliances of same technologies into networks reduces the number of appliances controllers and so it provides for larger scalability.

A large number of appliances in the house raise the question of performance, but given the small size of the SIP messages and the short transactions, network traffic has a little or no impact on this measurement. The performance may be affected by the reaction delay from the time the user agent receives the message to the time the appliance executes it. The user agent might take a small time to translate the action and send it to the specified device but it should not be a burden for the user.

Since this work was implemented and tested using a wired network, Ethernet, this presents a problem and a limitation in the home environment. Having wires running all over the house is not practical and it increases the equipment needed to implement it, thus, increasing the cost. Ideally, to take this implementation to the real world it should be via a wireless network which may introduce some security issues.

As the objectives of this work is concerned, the DO method was added to a SIP stack of an existing user agent, which could use this method to send requests via LAN to other user agents to perform specific actions. It is safe to say that this method is extensible to other SIP user agents with the modifications suggested in this work.

The objectives set for this research were successfully accomplished and it opens the door for future research on home automation using SIP. Some possible future work is discussed in the next section.

5.1 Future work

The study described in this document is just the starting point on using SIP to control and communicate home appliances. Future implementations of the DO method

may consider using the eXtensible Markup Language (XML) in the DO message body. Since there are a large number of home appliances, and each one of them is able to perform different actions, a specific XML schema can be defined to carry the requested action in the DO message body. The only component that needs to be added is an XML parser in the appliance controllers, this parser would let the controllers extract the action and pass it to the networked appliance.

Another option is defining a new protocol to communicate networked appliances. In order to do this, the message format needs to be defined, as well as the protocol primitives and its parameters.

Bibliography

[Camarillo2002]. Camarillo, G. 2002. SIP Demystified. McGraw-Hill TELECOM, New York.

[Moyer2000a]. Moyer, S. and Marples, D. 2000. The Internet Alarm Clock – A Networked Appliance Case Study. Telcordia Technologies.

[Moyer2000b]. Moyer, S. et. al. 2000. Service Portability of Networked Appliances. IEEE Communications Magazine.

[Moyer2001]. Moyer, S. Marples, D. and Tsang, S. 2001. A Protocol for Wide-Area Secure Networked Appliance Communication. IEEE Communications Magazine. October: 52-59.

[Rosenberg2002]. Rosenberg, J, et. al. 2002. SIP: Session Initiation Protocol. IETF RFC 3261.

[Sinnreich2001]. Sinnreich, H. and Johnson, A. 2001. Internet Communications Using SIP. Wiley Computer Publishing, New York.

[Tsang2000a]. Tsang, S., et. al. 2000. Requirements for Networked Appliances: Wide-Area Access, Control, and Interworking. IETF Internet Draft.

[Tsang2000b]. Tsang, S., et. al. 2000. SIP Extensions for Communicating with Networked Appliances. IETF Internet Draft.

[Wrolstad2003]. Wrolstad, J. 2003. IBM Helps Build Connected Community. http://www.wirelessnewsfactor.com/perl/story/21208.html.

Appendix A. Brief Overview of SIP Functionality

The Session Initiation Protocol (SIP) is a control protocol for the **application layer** that establishes, modifies, and terminates multimedia sessions such as Internet telephony calls [Rosenberg2002]. SIP works independently of the transport protocol and type of session established. It supports name mapping and redirection services, which supports personal mobility [Rosenberg2002]. SIP should be used in conjunction with other protocols (TCP/IP, SDP, among others) in order to provide complete services to the users. However, the basic functionality and operation of SIP does not depend on any of these protocols.

SIP provides primitives that may be used to implement different services. A single primitive is typically used to provide several services depending of the type of session being established. Here are the facets used to establish and terminating multimedia communications shown in [Rosenberg2002].

- User location: Determination of the end system to be used for communication.
- User availability: Determination of the willingness of the called party to engage in communications.
- User capabilities: determination of the media and media parameters to be used.
- Session setup: Establishment of session parameters at both parties.
- Session management: Including transfer and termination of sessions, modifying session parameters, and invoking services.

A complete description of SIP functionality and behavior can be found in IETF Requests for Comments (RFC) 3261, which is documented in [Rosenberg2002].

Appendix B. Pseudo Code

Application Layer Code

Global Data:

```
char action[512] = "\0";
char FROM[128] = "\0";
char TO[128] = "\0";
char REQUEST_URI[128] = "\0";
char *validActions[4] = {"Start Ringer", "Stop Ringer", "Backlight Turn on", "Backlight
Turn off"};
```

Function Definitions:

```
int setDoRequestMessage(char *user, char *to, char *from, char *doAction)
{
   sprintf(TO,  "To:sip:%s@%s", user, to);
   sprintf(FROM, "From:sip:%s@%s", g_sipControl->username, from);
   sprintf(REQUEST_URI, "sip:%s", to);
   strncpy(action, doAction, sizeof(action));
   return 0;
}

void AppSendDo()
{
   SipTranscHandle       hTransc; /*handle to the call-leg*/

   /*-------------------------
     creating a new Transaction
   ---------------------------*/
   status = SipTranscMgrCreateTransaction(g_sipControl->
                    transcMgr,NULL,&hTransc);
   if(status != SUCCESS)
   {
      printf("Failed to create new transaction");
   }
   printf("\ntransaction %x was created\n\n",(int)hTransc);

   /*-----------------------------------------------------------
    Send the request by calling the make function
   -------------------------------------------------------------*/
```

```
   printf("\nSending a %s request: \n\t%s -> %s\n\n",METHOD,FROM,TO);

   status = SipTransactionMake(hTransc,FROM,TO,REQUEST_URI,
         CSEQ,METHOD);
   if(status != SUCCESS)
   {
      printf("Transaction Make failed");
   }
}

void AppTranscStateChangedEvHandler(
                        IN  SipTranscHandle              hTransc,
                        IN  SipTranscOwnerHandle         hAppTransc,
                        IN  SipTransactionState          eState,
                        IN  SipTransactionStateChangeReason eReason)
{
   char methodStr[20];

   /*print the new state on screen*/
   printf("transc %x - State changed to %d\n\n", (int)hTransc, eState);

   switch(eState)
   {
      case SIP_TRANSC_STATE_SERVER_DO_REQUEST_RCVD:
      {
         SipTransactionGetMethodStr(hTransc,20,methodStr);
         if(strcmp(methodStr, "DO") == 0)
         {
            status = SipTransactionRespond(hTransc,
                  SIPCTRL_STATUS_OK, NULL);
         }
         else
         {
            status = SipTransactionRespond(hTransc,
                  SIPCTRL_STATUS_NOTIMPLEMENTED, NULL);
         }
         if(status != SUCCESS)
         {
             printf("Failed to respond to the request");
         }
      }
      break;
```

```
   default:
      break;
   }
}

void AppTranscMsgReceivedEvHandler(
                           IN  SipTranscHandle          hTransc,
                           IN  SipTranscOwnerHandle     hAppTransc,
                           IN  SipMsgHandle             hMsg)
{
    SipMethodType method;
   unsigned int bodyLength;
   int bodyBuffer = 256;
   char message[bodyBuffer];
   int actualLength;

   printf("<-- Message Received (transaction %x)\n", (int)hTransc);
   /* Get method from message */
   method = SipMsgGetRequestMethod(hMsg);
   switch(method)
   {
      case SIP_METHOD_DO:
      {
         /* Get body message length */
         bodyLength = SipMsgGetContentLengthHeader(hMsg);
         if(bodyLength >= bodyBuffer)
         {
            bodyBuffer = bodyLength;
         }

         /* Get message */
         SipMsgGetBody(hMsg, message, bodyBuffer, &actualLength);
         printf("DO Action message: \n %s\n", message);
         if(isValidAction(message))
         {
            executeAction(message);
         }
      }
      break;
      default:
         break;
   }
```

```
}

void AppTranscMsgToSendEvHandler(
                        IN  SipTranscHandle         hTransc,
                        IN  SipTranscOwnerHandle    hAppTransc,
                        IN  SipMsgHandle            hMsg)
{
        char messageBuf[512];
        char content[128];

        /* Set Content Type header */
        sprintf(content, "application/text");
        SipMsgSetContentTypeHeader(hMsg, content);
        /* Set Body message with specified action */
        sprintf(messageBuf, "%s", action);
        SipMsgSetBody(hMsg, messageBuf);

        printf(logstr, "--> Message Sent transc %x\n", (int)hTransc);
}

bool isValidAction(char *action)
{
   int i;

   for(i = 0; i < 4; i++)
   {
      if(strcmp(action, validActions[i]) == 0)
      {
         return true;
      }
   }
   return false;
}

void executeAction(char *action)
{
   if(strcmp(action, "Start Ringer") == 0)
   {
      uiu_set_ring(true);
      return;
   }
   if(strcmp(action, "Stop Ringer") == 0)
```

```
    {
        uiu_set_ring(false);
        return;
    }
    if(strcmp(action, "Backlight turn on") == 0)
    {
        coBackLight(true);
        return;
    }
    if(strcmp(action, "Backlight turn off") == 0)
    {
        coBackLight(false);
        return;
    }
    return;
}
```

Control Layer Code

Declarations:

```
typedef enum
{
    SIP_METHOD_UNDEFINED = -1, /* undefined method type. */
    SIP_METHOD_INVITE,        /* user or service is being invited to participate
                                                    in a session. */
    SIP_METHOD_ACK,          /* confirmation that the client has received a final
                                                    response to INVITE request. */
    SIP_METHOD_BYE,          /* The user agent uses BYE to indicate to the server that
                                                    it wishes to release the call. */
    SIP_METHOD_REGISTER,      /* The user agent client requests to register
                            to a registrar server by the addresses sent
                            in the Contact header of the REGISTER request*/
    SIP_METHOD_REFER,         /* Triggers the server to initiate a call with a
                          * third party. */
    SIP_METHOD_NOTIFY,        /* Notifies of an event occurring. */
    SIP_METHOD_DO,                /* Sends an action to be performed */
    SIP_METHOD_OTHER,         /* not one of the above */
    SIP_METHOD_PRACK,
    SIP_METHOD_CANCEL

} SipMethodType;
```

```
typedef enum
{

   PARSER_METHOD_TYPE_INVITE,    /* user or service is being invited to
participate in a session. */
   PARSER_METHOD_TYPE_ACK,       /* confirmation that the client has received a
final response to INVITE request. */
   PARSER_METHOD_TYPE_BYE,       /* The user agent uses BYE to indicate to the
server that it wishes to release the call. */
   PARSER_METHOD_TYPE_OPTIONS,   /* The server is being queried as to its
capabilities */
   PARSER_METHOD_TYPE_CANCEL,    /* The CANCEL request cancels a pending
request with the same Call-ID, To, From and CSeq (sequence number only)
header field values, but does not affect a completed request or existing calls. (A request is
considered completed if the server has returned a final status response.)*/
   PARSER_METHOD_TYPE_REGISTER,  /* A client uses the REGISTER method to
bind the address listed in the To header field with a SIP server to one or more URIs
where the client can be reached, contained in the Contact header fields. These URIs may
use any URI scheme, not limited to SIP.*/
   PARSER_METHOD_TYPE_REFER,
   PARSER_METHOD_TYPE_NOTIFY,
   PARSER_METHOD_TYPE_DO,
   PARSER_METHOD_TYPE_PRACK,
   PARSER_METHOD_TYPE_INFO,
   PARSER_METHOD_TYPE_SUBSCRIBE,
   PARSER_METHOD_TYPE_OTHER      /* none of the above */
} ParserMethodType;
```

Transaction Layer Code

Declarations:

```
typedef enum
{
   SIP_TRANSACTION_METHOD_UNDEFINED = -1,
   SIP_TRANSACTION_METHOD_INVITE,
   SIP_TRANSACTION_METHOD_BYE,
   SIP_TRANSACTION_METHOD_REGISTER,
   SIP_TRANSACTION_METHOD_REFER,
   SIP_TRANSACTION_METHOD_NOTIFY,
   SIP_TRANSACTION_METHOD_DO,
```

```
   SIP_TRANSACTION_METHOD_CANCEL,
   SIP_TRANSACTION_METHOD_PRACK,
   SIP_TRANSACTION_METHOD_OTHER

} SipTransactionMethod;

typedef enum
{
   SIP_TRANSC_STATE_UNDEFINED = -1,
   SIP_TRANSC_STATE_IDLE,
   SIP_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD,
   SIP_TRANSC_STATE_SERVER_GEN_FINAL_RESPONSE_SENT,
   SIP_TRANSC_STATE_CLIENT_GEN_REQUEST_SENT,
   SIP_TRANSC_STATE_CLIENT_GEN_PROCEEDING,
   SIP_TRANSC_STATE_CLIENT_INVITE_CALLING,
   SIP_TRANSC_STATE_CLIENT_INVITE_PROCEEDING,
   SIP_TRANSC_STATE_CLIENT_INVITE_FINAL_RESPONSE_RCVD,
   SIP_TRANSC_STATE_CLIENT_INVITE_ACK_SENT,
   SIP_TRANSC_STATE_SERVER_INVITE_REQUEST_RCVD,
   SIP_TRANSC_STATE_SERVER_INVITE_FINAL_RESPONSE_SENT,
   SIP_TRANSC_STATE_TERMINATED,
   SIP_TRANSC_STATE_SERVER_PRACK_FINAL_RESPONSE_SENT,
   SIP_TRANSC_STATE_SERVER_INVITE_REL_PROV_RESPONSE_SENT,
   SIP_TRANSC_STATE_SERVER_INVITE_PRACK_COMPLETED,
   SIP_TRANSC_STATE_CLIENT_INVITE_PROXY_2XX_RESPONSE_RCVD,
   SIP_TRANSC_STATE_SERVER_INVITE_PROXY_2XX_RESPONSE_SENT,
   SIP_TRANSC_STATE_SERVER_CANCEL_REQUEST_RCVD,
   SIP_TRANSC_STATE_CLIENT_INVITE_CANCELLING,
   SIP_TRANSC_STATE_CLIENT_GEN_CANCELLING,
   SIP_TRANSC_STATE_CLIENT_CANCEL_SENT,
   SIP_TRANSC_STATE_CLIENT_CANCEL_PROCEEDING,
   SIP_TRANSC_STATE_SERVER_CANCEL_FINAL_RESPONSE_SENT,
   SIP_TRANSC_STATE_CLIENT_GEN_FINAL_RESPONSE_RCVD,
   SIP_TRANSC_STATE_CLIENT_INVITE_PROCEEDING_TIMEOUT,
   SIP_TRANSC_STATE_SERVER_INVITE_ACK_RCVD,
   SIP_TRANSC_STATE_CLIENT_CANCEL_FINAL_RESPONSE_RCVD,
   SIP_TRANSC_STATE_CLIENT_DO_REQUEST_RCVD,
   SIP_TRANSC_STATE_CLIENT_DO_REQUEST_SENT
} SipTransactionState;
```

Function Definitions:

```
Status TransactionMsgReceived(IN Transaction   *pTransaction,
                              IN RvSipMsgHandle hMsg)
{
   SipMsgType       bMsgType;
   Status           retStatus;

   bMsgType = SipMsgGetMsgType(hMsg);
   if (SIP_MSG_REQUEST == bMsgType)
   {
      /*The message is a request message*/
      retStatus = HandleRequestMsg(pTransaction, hMsg);
      if ((Success != retStatus) && (SIP_TRANSC_STATE_IDLE == pTransaction->
         eState))
      {
         LOG_Print (pTransaction->pMgr->hLogHandle, ERROR,(pTransaction->pMgr->
                  hLogHandle, ERROR, "TransactionMsgReceived -Failed for transaction
                  %x - terminating with error."));
         TransactionTerminate(SIP_TRANSC_REASON_ERROR, pTransaction);
      }
      return retStatus;
      }
      else if (SIP_MSG_RESPONSE == bMsgType)
      {
         /*The message is a response message*/
         UINT16 responseCode;
         responseCode = SipMsgGetStatusCode(hMsg);
         if ((100 <= responseCode) && (responseCode < 200))
         {
            /* Provisional response */
            retStatus = HandleProvisionalResponse(pTransaction, hMsg);
            return retStatus;
         }
         else if ((200 <= responseCode) && (700 > responseCode))
         {
            /* Final response */
            retStatus = HandleFinalResponse(pTransaction, hMsg);
            return retStatus;
         }
         else
         {
```

```
            /* Response code is out of range */
            return InvalidParameter;
        }
    }
    /*The message is not a request or a response message*/
    return InvalidParameter;
}

Status HandleRequestMsg(IN Transaction   *pTransaction,
                                                IN RvSipMsgHandle hMsg)
{
    Status       retStatus;
    SipMethodType eMethod;

    switch (pTransaction->eState)
    {
        case (SIP_TRANSC_STATE_IDLE):
            retStatus = HandleRequestInInitialState(pTransaction, hMsg);
            if (Success != retStatus)
            {
                return retStatus;
            }
            break;
        case (SIP_TRANSC_STATE_SERVER_INVITE_FINAL_RESPONSE_SENT):
            eMethod = SipMsgGetRequestMethod(hMsg);
            if (SIP_METHOD_ACK == eMethod)
            {
                /* Ack method received in a server Invite in the "Final Response Sent state */
                LOG_Print(pTransaction->pMgr->hLogHandle ,INFO, (pTransaction->pMgr->
            hLogHandle,INFO, "TransactionRequestMsgReceived - Transaction %x: has
            received Ack request", pTransaction));


            /* Call "message received" callback */
            if (NULL != (pTransaction->pEvHandlers)->pfnEvMsgReceived)
            {
                (pTransaction->pEvHandlers)->pfnEvMsgReceived(
                                (RvSipTranscHandle)pTransaction,
                                pTransaction->pOwner,
                                hMsg);
            }
            retStatus =   TransactionStateServerInviteAckRcvd(
```

```
                        SIP_TRANSC_REASON_ACK_RECEIVED,
                        pTransaction);

          if(retStatus != Success)
          {
             return retStatus;
          }
        }
         break;
         case SIP_TRANSC_STATE_SERVER_INVITE_REQUEST_RCVD:

          if (NULL != (pTransaction->pEvHandlers)->pfnEvMsgReceived)
          {
             (pTransaction->pEvHandlers)->pfnEvMsgReceived(
                          (SipTranscHandle)pTransaction,
                          pTransaction->pOwner,
                          hMsg);
          }
        break;
        case SIP_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD:
        case SIP_TRANSC_STATE_SERVER_CANCEL_REQUEST_RCVD:
          /* Retransmit the last provisional response message, if exists */
          LOG_Print((pTransaction->pMgr)->hLogHandle ,INFO, ((pTransaction->pMgr)-
>hLogHandle,INFO, "TransactionRequestMsgReceived - Transaction %x: has received
request retransmission", pTransaction));
          retStatus = TransactionTransportRetransmitMessage(pTransaction);
          if (Success != retStatus)
          {
             return retStatus;
          }
         break;
        case (SIP_TRANSC_STATE_SERVER_GEN_FINAL_RESPONSE_SENT):
        case (SIP_TRANSC_STATE_SERVER_CANCEL_FINAL_RESPONSE_SENT):
        case (SIP_TRANSC_STATE_SERVER_PRACK_FINAL_RESPONSE_SENT):
          /* Retransmit the final response message */
          LOG_Print((pTransaction->pMgr)->hLogHandle ,INFO,
               ((pTransaction->pMgr)->hLogHandle,INFO,
               "TransactionRequestMsgReceived - Transaction %x: has received request
retransmission", pTransaction));
                    retStatus = TransactionTransportRetransmitMessage(pTransaction);
          if (Success != retStatus)
          {
```

```
            return retStatus;
          }
        break;
          /* Proxy received ACK after sending 2xx response or an
            ACK retransmission*/
      case
(SIP_TRANSC_STATE_SERVER_INVITE_PROXY_2XX_RESPONSE_SENT):
        {
           eMethod = SipMsgGetRequestMethod(hMsg);
          if(SIP_METHOD_INVITE == eMethod)
          {
            LOG_Print((pTransaction->pMgr)->hLogHandle ,INFO,
                        ((pTransaction->pMgr)->hLogHandle,INFO,
                        "TransactionRequestMsgReceived - A proxy Transc=%x: received
request retransmission msg=%x", pTransaction,hMsg));
            return(Success);
          }
          else
          {
            LOG_Print((pTransaction->pMgr)->hLogHandle ,RV_EXCEP,
                          ((pTransaction->pMgr)->hLogHandle,RV_EXCEP,
                          "TransactionRequestMsgReceived - A proxy Transc=%x:
received request=%d on State:Proxy2XXSent msg=%x", pTransaction,eMethod, hMsg));
            return(IllegalAction);
          }
        }
        break;
        default:
           LOG_Print((pTransaction->pMgr)->hLogHandle ,INFO, ((pTransaction->pMgr)-
>hLogHandle,INFO, "TransactionRequestMsgReceived - Transaction %x: received a
request and ignored it", pTransaction));
        break;
        }
    return Success;
}

Status HandleRequestInInitialState(IN Transaction   *pTransaction,
                           IN RvSipMsgHandle hMsg)
{
    SipMethodType eMethod;
    Status       ret;
```

```
   eMethod = SipMsgGetRequestMethod(hMsg);
   if(eMethod != SIP_METHOD_ACK && eMethod != SIP_METHOD_CANCEL &&
     pTransaction->pMgr->rejectUnsupportedExtensions == RV_TRUE)
   {
      Status ret;
      ret = TransactionSetUnsupportedList(pTransaction,hMsg);
      if(ret != Success)
      {
         return ret;
      }
   }

   /* set authorization headers list from msg in the transaction -
     not for cancel and ack, because you can't challenge this kind of requests (rfc, bis6) */
   if(TRUE == pTransaction->pMgr->enableServerAuth)
   {
      if((SIP_METHOD_CANCEL != eMethod) && (RVSIP_METHOD_ACK !=
eMethod))
      {
         ret = TransactionAuthSetAuthorizationList(pTransaction, hMsg);
         if(ret != Success)
         {
            LOG_Print((pTransaction->pMgr)->hLogHandle ,RV_ERROR,
((pTransaction->pMgr)->hLogHandle, RV_ERROR,
                  "TransactionRequestReceivedInInitialState - Transaction %x: Error - fail to
set authorization list",
                  pTransaction));
            return ret;
         }
      }
   }

   /*copy the time stamp to the transaction page*/
   ret = TransactionCopyTimestampFromMsg(pTransaction,hMsg);
   if(ret != Success)
   {
      return ret;
   }

   switch (eMethod)
   {
   case (SIP_METHOD_INVITE):
```

```
      /* The request received is an Invite request */
      return HandleDoInInitialState(pTransaction, hMsg);
   case (SIP_METHOD_ACK):
      /* Ack Request has been received in the Initial state */
      LOG_Print((pTransaction->pMgr)->hLogHandle ,EXCEP,
            ((pTransaction->pMgr)->hLogHandle, EXCEP,
            "TransactionRequestReceivedInInitialState - Transaction %x: Error - A new
server transaction received Ack request",
            pTransaction));
      return InvalidParameter;
   case (SIP_METHOD_BYE):
      /* The request received is a BYE request */
   case (SIP_METHOD_REGISTER):
      /* The request received is a REGISTER request */
   case (SIP_METHOD_REFER):
      /* The request received is a REFER request */
   case (SIP_METHOD_NOTIFY):
      /* The request received is a NOTIFY request */
   case (SIP_METHOD_OTHER):
   case (SIP_METHOD_CANCEL):
      /* The request received is a NOTIFY request */
      return HandleCancelInInitialState(
                  pTransaction, hMsg, SIP_TRANSACTION_METHOD_CANCEL);
   case (SIP_METHOD_DO):
        /* The request received is a DO request */
      return HandleDoInInitialState(pTransaction, hMsg);
   case (SIP_METHOD_PRACK):
      /* The request received is a PRACK request, if the transaction is a proxy transaction
call general recvd insted only if the proxy didn't send the 1xx reliable.*/
      if(TRUE == pTransaction->bIsProxy)
      {
         Transaction* pPrackPair= NULL;
         /*find the Invite transaction*/
         TransactionMgrHashFindMatchPrackTransc(pTransaction->pMgr,
                        (SipTranscHandle)pTransaction,
                        hMsg,
                        (SipTranscHandle*)&pPrackPair);
         if((NULL != pPrackPair) &&
(SIP_TRANSC_STATE_SERVER_INVITE_REL_PROV_RESPONSE_SENT==
pPrackPair->eState))
         {
            return HandlePrackInInitialState(pTransaction, hMsg);
```

```
            }
            else
            {
                return HandleGeneralInInitialState(pTransaction, hMsg);
            }
        }
        else
        {
            return HandlePrackInInitialState(pTransaction, hMsg);
        }
    default:
        LOG_Print((pTransaction->pMgr)->hLogHandle ,EXCEP,
                ((pTransaction->pMgr)->hLogHandle,EXCEP,
                "TransactionRequestReceivedInInitialState - Transaction %x: Error - A new
server transaction received request with undefined method",
                pTransaction));
        return RV_InvalidParameter;
    }
}

Status HandleDoInInitialState(IN Transaction    *pTransaction,
                        IN RvSipMsgHandle hMsg)
{
    Status retStatus;
    SipMethodType method;
    unsigned int bodyLength;
    int bodyBuffer = 256;
    char message[bodyBuffer];
    int actualLength;
    Status status;
    char logstr[512];

    /* Call "message received" callback */
    if (NULL != (pTransaction->pEvHandlers)->pfnEvMsgReceived)
    {
        (pTransaction->pEvHandlers)->pfnEvMsgReceived(
                                    (SipTranscHandle)pTransaction,
                                    pTransaction->pOwner,
                                    hMsg);
    }
   /* Set the transaction's method to DO */
    bodyLength = SipMsgGetContentLengthHeader(hMsg);
```

```
   if(bodyLength >= bodyBuffer)
   {
      bodyBuffer = bodyLength;
   }
   /* Get message */
   status = SipMsgGetBody(hMsg, message, bodyBuffer, &actualLength);
   printf("DO Action message: \n %s\n", message);
   pTransaction->eMethod = SIP_TRANSACTION_METHOD_DO;

   /* Add To Tag if required */
   /* Copy the list of Via headers from the message to the transaction */
   retStatus = TransactionUpDateViaList(pTransaction, hMsg);
   if (Success != retStatus)
   {
      LOG_Print((pTransaction->pMgr)->hLogHandle ,RV_ERROR,
            ((pTransaction->pMgr)->hLogHandle,RV_ERROR,
                        "TransactionInviteReceievdInInitialState - Transaction %x:
Error in trying to construct a new Via list.",
            pTransaction));
      return retStatus;
   }
   /* The transaction assumes "Do Request Received" state */
   retStatus = TransactionStateDoReqRecvd(
                        SIP_TRANSC_REASON_REQUEST_RECEIVED,
                        pTransaction);

   RvSipTransactionRespond((SipTranscHandle)pTransaction,200,NULL);

   if (Success != retStatus)
   {
      return retStatus;
   }

  return Success;
}

Status TransactionStateDoReqRecvd(
                        IN RvSipTransactionStateChangeReason eReason,
                        IN Transaction *pTransaction)
{
   /* Release timer */
         TransactionTimerRelease(pTransaction);
```

```
    TransactionChangeState(pTransaction,
                    SIP_TRANSC_STATE_CLIENT_DO_REQUEST_RCVD,
                    eReason,
                    NULL);

    return Success;
}
```