The Network Transport and Performance (NTAP) Test Bed for WebRTC applications

Elias Yousef
A20302787
06/07/2014
Abstract

This paper discusses the WebRTC NAT Traversal problem, the Implementation of STUN and TURN in the Test bed is particularly explained in this report. Then, the test bed developed is presented to test the performance of WebRTC-based applications. The focus here was testing the WebRTC Signaling Performance and Turn server Performance.
# Table of Contents

## Contents
The Network Transport and Performance (NTAP) Test Bed for WebRTC applications...........................................................................................................1
Abstract..........................................................................................................................................................................................2
Table of Contents..............................................................................................................................................................3
1. Introduction : .................................................................................................................................................................4
   *The WebRTC Based Application* .................................................................................................................................11
3. The Network Traversal and Performance Test-Bed........................................................................................................12
4. Scenarios and Measurements : .......................................................................................................................................15
Conclusions .................................................................................................................. Error! Bookmark not defined.
References ........................................................................................................................................................................17
Appendices ........................................................................................................................................................................20
1. Introduction:
Most of nowadays households/Companies reach the internet using (NAT). Traditionally people connect to the internet directly, However there are not enough addresses for every device there. As we know, the NAT (Network Address Translator) allows us to connect to the internet using a single Address (Public IP address) and share multiple devices, However This causes a huge problem in nowadays WebRTC applications.

2. Preliminaries:

2.1 WebRTC:
Definition: The WebRTC as defined in [1]: “WebRTC is an industry and standards effort to put real-time communications capabilities into all browsers and make these capabilities accessible to web developers via standard [HTML5] tags and JavaScript APIs (Application Programming Interfaces).”

2.2 RTC-function in the browser:

![RTC Function In the Browser](image)

Figure -1- RTC Function In the Browser
2.3 WebRTC basic Architecture:
Two main architectures are used to implement WebRTC, I will briefly introduce each one of them. The main architecture used in this project is the Triangle.

2.3.a - WebRTC Triangle:
This is the simplest and most common scenario were both browsers for users are running the same WebRTC application from the same webpage on the webserver that Produces a Triangle as shown in the figure were the sides of the triangle are the signaling and the base of the triangle is the media Relay.

![WebRTC Triangle Architecture](image)

2.3.b - WebRTC Trapezoid:
The two webservers will exchange signaling through common protocol like SIP, each browser will communicate with the associated server before establishing the peer to peer connection and exchanging media as shown in figure.

![WebRTC Trapezoid Architecture](image)
2.4 -NAT Problem:

2.4.1-Introduction to Nat Problem:
in a classical old scenario: When there is an incoming call going to the destination party, it could reach the device directly if it is connected directly to the internet. But the existence of the NAT will the corresponding call will go through the internet and reach the destination NAT and The latter cant figure out which device the call should be handed off, and as a result it drops the call.
The user that initiates this call will see that at times the call will be initiated and other times it is not. The solution that makes this call go through was STUN (simple Traversal Of UDP Through NAT) primarily used for residential NAT. STUN allows two devices behind NAT to have a peer to peer connection through the NAT.

![Diagram](image.png)

Figure -4- WebRTC users Behind NAT(Need of Stun)

The stun solution does not solve the problem 100% reliable, sometimes we have Enterprise NATs that will have configuration that will not allow UDP traffic to go through where all the RTC Media traffic is carried on. In the case of the Enterprise NAT. There should be a Relay which relates to both parties addresses. This relay is achieved by a server that is Called a TURN (Traversals Using Relay NAT) server.
Figure 5- WebRTC users Behind NAT and Firewalls (Need of Turn)

The problem doesn’t end here. Using the TURN Server as a media relay consumes a lot of bandwidth. The best solution would be using the STUN server whenever possible, and if not, use the Turn server. The ICE (Interactive Connectivity Establishment) framework implement that Methodology and is used in our project to maximize reliability of the webRTC applications in nowadays more complex NAT structures.

Figure 6- WebRTC users And ICE
2.4.2-Server reflexive address:
A NAT device works by associating a public address and port with a private destination address and port.

<table>
<thead>
<tr>
<th>Public</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>206.123.31.67:55123</td>
<td>↔ 192.168.1.2:5060</td>
</tr>
</tbody>
</table>

The public address and port together are known as the server-reflexive address. For the majority of NAT devices (mostly home routers), any device on the Internet may contact the NATed party by sending packets to the server-reflexive address, even if they are not the receiver of the connection-initiating packet. A mean for discovering the server-reflexive address and communicating it to the other party is therefore needed. Here comes the importance of STUN server.

2.4.3-STUN:
Session Traversal Utilities for NAT (STUN) is a simple protocol for discovering the server-reflexive address. The NATed peer initiates a connection to the STUN server, thus creating a binding in the NAT device. The STUN server receives the query and inspects the sender address, which is the server-reflexive address. It sends back a reply containing the server-reflexive address in its payload. The client thus learns its server-reflexive address. The Flow Diagram of the Stun operation is explained in the following ladder:
2.4.4-*STUN 2*:
It turns out that some NAT devices try to be clever by inspecting the payloads and changing all references to the server-reflexive address into the private address. To address that issue, the new version of STUN (known as STUN 2, still an IETF draft) obfuscates the address by XORing it with a known value.

2.4.5-Symmetric NAT:
Some NAT devices only allow packets from the remote peer to reach the NATed peer. Thus a STUN request is useless because only the STUN server could reach the NATed peer through the server-reflexive address. These NAT devices are called symmetric NATs. They are often “enterprise” NATs that hide more devices on average. Thus, their presence is significant and must be worked around.

2.4.6-*TURN*:
To be reachable, a device behind a symmetric NAT needs to initiate and maintain a connection to a relay. Traversal Using Relays around NAT (TURN) is a protocol for communicating with the relay. It is built on top of STUN. The TURN server is located outside the NAT, either on the public Internet or in an ISP’s network when offered as a service by the ISP. A NATed TURN client asks the server to allocate a public address and port and relay packets to and from that address.

2.4.7-*TURN and Relayed address*:
The address allocated by the TURN server is called the relayed address. The TURN server communicates that address to the TURN client. TURN guarantees communication in all NAT cases unless there is an explicit firewall policy to prohibit its use. The Flow Diagram of TURN protocol is illustrated next:
Turn has major problems: it requires a lot of bandwidth. Server must remain available for the whole duration of the allocation. Additional headers consume a bit more bandwidth.

2.4.8 ICE:
Interactive connectivity establishment is a protocol that solves the TURN load problem, it will try to use Stun server first if possible, and use the TURN server as a last result, so we avoid using Turn with ICE. ICE tries to connect peers directly via UDP. If UDP fails, ICE tries TCP: first HTTP, then HTTPS. If direct connection fails—in particular, because of enterprise NAT traversal and firewalls—ICE uses an intermediary (relay) TURN server.

2.5 - Call flow and JavaScript session establishment of WebRTC triangle:
The JSEP is a new signaling architecture developed for WebRTC [1, p. 95]. It is not a signaling protocol like SIP or Jingle, but it explains how a Javascript application running on a browser can interact with the RTC function. It describes how the Javascript can obtain information about the capabilities of the browser (codecs and media supported for example). It also explain the offer/answer mechanism used by the browsers to negotiate media configuration, and the Interactive Connectivity Establishment (ICE) hole punching process. JSEP uses SDP for the offer and the answer syntaxes.

Call flow
Two clients, M and L, are connected to a Web server that handle WebRTC. First of all, they retrieve the web page of the application thanks to a HTTP GET request. When M tries to call L, M will first create a SDP message. This SDP message contains important information about the client, including the supported codecs and an IP address where it is reachable. This IP address can be private or public. This message is called an offer. This offer is sent by M to L, via the web server. When L receives this message, it creates an SDP answer, on the same model as the offer, and sends it to M via the web server. When this exchange is finished, they still continue to send each other some messages via the web server, called Interactive Connectivity Establishment (ICE) messages. Those messages contains IP addresses (obtained from a STUN for example) and ports where they are reachable. Indeed, a client may have different IP addresses because of the presence of a NAT or because it has multiple network interfaces. Meanwhile, they begin a mechanism called hole punching, which consists in sending STUN requests to the other clients, using the IP addresses contained in the SDP message and the ICE messages. When they reach each other, they begin the key negotiation for the media session and then communicate, in a secure way, on the PeerConnection, with a secure media session.
The WebRTC Based Application

This application is a simple Audio/Video chat between two clients. It uses a triangle architecture. The web page was inspired by the W3C PeerConnection example [2]. This application works as explained in part 2.5. For the signaling, a WebSocket server is used. NodeJS is the software used for the implementation of the WebSocket server. The code of this application (HTML and signaling) is presented in Appendix 3.
3. The Network Traversal and Performance Test-Bed

Many questions face WebRTC application developers when it comes to complex network architecture over the internet.

Will the application work in Enterprise network architecture?
What architecture of testbed should we have?
Where should the location of Stun/Turn server be?

The Network Traversal and Performance (NTAP) Test-bed was built to test WebRTC applications under different network configurations that are similar to the ones in complex networks.

3.1 - The logical architecture: In our Lab, we implemented a testbed that could create very close to real life scenarios where Users are located in Different domains and may have Private IP addresses laying behind Symmetric NATs. Each domain is represented by a router.

We use seven domains that are represented by eight routers, each router had its unique configuration that we change according to the scenario, the most important configuration are the access lists that represent the firewalls and NATS. Router Configurations are shown in Appendix 2.

We tried to make the testbed Multi layered to be able to make scenarios were the call should go through different layers of NATS and translations from private to public addresses. A turn/Stun server explained in [3] has been installed on linux machine and located to do tests on different domains.

As demonstrated in the logical architecture below the architecture has three major domains on the top, Domain 1,2, and 3. That are all connected to a switch, and each domain will have sub-domains related to it. For example Domain one will have subdomain that will be demonstrated in the figure below as the second layer and will have the name D11 since it is a sub-domain from domain one (see naming convention for more details about naming domains and devices/users).
3.2 - The physical architecture:

Each router that represents a single domain is connected to a hub. Other sub Domains, higher Domains, User agents, and servers are connected to the corresponding domain’s hub. The Switch that connects the top three domains could be connected to other test-beds like NG911 test-bed that will extend the possibility of testing applications. It also could be connected to the internet to make more options possible.
The different routers, hubs, Switches and devices constitute the physical architecture for the NTAP testbed that is represented in the physical architecture below:

Figure -9- Network Traversal and Performance test-bed Logical Architecture
3.3 – presentation and naming convention:

Naming Convention used for the Domains is based on the rules:
1– the highest Domain in the Testbed gets one digit, For example in our Test-bed, D1,D2 and D3.
2- the domain that is a subdomain(lower level) from a higher level domain, takes the previous domain digits and adds one digit to it, for example the second layer domains D11, D21.
3 – any devices connected to These domains, will add a letter after the domain name to be distinguished from domain name, for example, D21A is a computer connected to domain D21.

More information about the Test-bed configurations are available in the appendix 1

4. Scenarios and Measurements:

One of the main things that we focus on when testing WebRTC applications is peer-connection session establishment with ICE protocol. This simple WebRTC audio/Video Chat has been tested through many scenarios where the application on different Clients Creates a peer connection. Results for different scenarios are shown in the following table:

<table>
<thead>
<tr>
<th>Case</th>
<th>Turn Location</th>
<th>From</th>
<th>To</th>
<th>No-Ice</th>
<th>Stun</th>
<th>Turn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D1</td>
<td>D1</td>
<td>D31</td>
<td>NO</td>
<td>Works</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>D1</td>
<td>D1</td>
<td>D211</td>
<td>NO</td>
<td>Works</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>D1</td>
<td>D21</td>
<td>D211</td>
<td>Yes</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>4</td>
<td>D1</td>
<td>D211</td>
<td>D311</td>
<td>NO</td>
<td>NO</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>D1</td>
<td>D21</td>
<td>D31</td>
<td>NO</td>
<td>NO</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>D11</td>
<td>D1</td>
<td>D31</td>
<td>NO</td>
<td>YES</td>
<td>NT</td>
</tr>
<tr>
<td>7</td>
<td>D11</td>
<td>D1</td>
<td>D211</td>
<td>Yes</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>8</td>
<td>D11</td>
<td>D21</td>
<td>D31</td>
<td>NO</td>
<td>Yes</td>
<td>NT</td>
</tr>
<tr>
<td>9</td>
<td>D11</td>
<td>D21</td>
<td>D211</td>
<td>Yes</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>10</td>
<td>D11</td>
<td>D211</td>
<td>D311</td>
<td>NO</td>
<td>No</td>
<td>yes</td>
</tr>
<tr>
<td>11</td>
<td>D11</td>
<td>D211</td>
<td>D311</td>
<td>NO</td>
<td>YES</td>
<td>NT</td>
</tr>
<tr>
<td>12</td>
<td>D0</td>
<td>D211</td>
<td>D311</td>
<td>NO</td>
<td>NO</td>
<td>yes</td>
</tr>
</tbody>
</table>
Explanation of Columns:

Case : number of scenario
Turn Location : the location of turnserver using Architecture I or II
From : Domain at which user agent 1 is located at
To : Domain at which user agent 2 is located at
No-ICE : does the call work without Stun/Turn ? (NO ice Protocol)
Stun : Does the call succeed in case we implement Stun ?
Turn : Does the call succeed in case we implement Turn?

Cases 3,9: In this case, a client belongs to a domain, and the other belongs to the subdomain relative to the as previously. Because of the NAT, Client1 will see Client2 with an IP of his own network. Indeed, a symmetric NAT is used.

Cases 1,2,5,6,8 : clients are located in different domains behind a NAT so basically, they can not use their private IP to communicate. That is why the connection fails without STUN/TURN. By using the STUN server, each client can retrieve his own public IP address and share it with the other peer thanks to the ICE candidate. Finally, both can reach other thanks to the public IP address.

Cases 4,10: the difference is the presence of a restrictive firewall on Routers F and G. RouterG forbids UDP data coming from RouterF (D211) network. RouterF forbids UDP data coming from RouterG (D311) network. As a result, although each client know each other public IP address, they are not allowed to Communicate. In this case, the TURN server acts again like a media relay between the clients, using UDP.

Cases 11,12: in this case we repeat case 10 but after removing the firewall restrictions on routers F and G, then in this case the Stun server will work as in the cases 1,2,5,6 and 8. In 12 the scenario is Similar to 11 but the location of the Turn server is on D0 which is a higher level domain, comparing wireshark traces shows that trip time dor a packet from User Agent 1 to User Agent 2 in 12 is shorter that this in 11. This is because the Decreased number of hops by 2.

5.Performance Measurements:

5.1 – Signaling Performance:
To test the performance of signaling a socket.IO server has been installed on the Server where the signaling process implemented on [4] is shown in the following Figure where SDP messages are carried by websocket socket.io. the Java scripts that are run by Node.JS are shown in the Appendix 4
As mentioned in [4] the transaction or signaling steps are:
- The client opens a socket on the server
- The client sends a SDP offer (hard coded) to the server, with socket.io
- The server processes the request by looking in his users table to which socket he has to respond
- The server transmits the SDP offer to the same client
- The client sends the SDP answer to the server
- The server processes the request by looking in his users table to which socket he has to respond
- The server transmits the SDP answer to the same client
- The client closes the connection
This process takes about 16 ms to execute.
The tests that was done in 4 were the capacity of the server to handle multiple transactions. Here, we will test the server’s CPU and memory performance when number of Threads is increasing, The server is a Linux Centos 6.4, 4GB Memory and intel dual core 2.5 GHZ processor. The delay between transactions was 20 ms the results are shown in figure 11.
After a while, the server is overloaded and cannot process more clients when clients are added or delay is reduced. The main reason is the fact that the clients connect so quickly that the server does not have the time to compute them all because of the increasing usage of server’s CPU. Also, as some clients don't correctly disconnect, the server keeps open connections for non-existent clients. So, at some point as we see in the figure when number of threads reach 5562, new clients cannot connect because the server is full of connected zombie clients. The signaling server is efficient when it has to compute requests linearly, but it quickly fails when parallel users are added.

5.2 – Turn Server Performance :

The performance of the turn server has been tested when using a conference application (BABL) that was developed in the RTC Lab. All users has been located in domains that have firewall restrictions so they are forced to use the Turnserver as a media relay. The Usage of the CPU from the Turnserver process has been monitored each time a user joins the conference call, The results are shown in the figure 12, The TurnServer is a Linux Centos 6.4, 4GB Memory and intel dual core 2.5 GHZ processor. A grade from 1 to 5 has been given by user 1 that is using a windows 7 core I5 4 GB computer.
<table>
<thead>
<tr>
<th>User Number</th>
<th>Turn CPU Usage %</th>
<th>QOE</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Users</td>
<td>0</td>
<td>N/A</td>
<td>0.1% Mem Usage</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0.1% Mem Usage</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0.1% Mem Usage</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>5</td>
<td>0.1% Mem Usage</td>
</tr>
<tr>
<td>5</td>
<td>2.3</td>
<td>4</td>
<td>0.1% Mem Usage</td>
</tr>
<tr>
<td>6</td>
<td>2.7</td>
<td>4</td>
<td>0.1% Mem Usage</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>4</td>
<td>0.1% Mem Usage</td>
</tr>
<tr>
<td>7</td>
<td>3.3</td>
<td>3</td>
<td>Lots of motion</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>0.1% Mem Usage</td>
</tr>
</tbody>
</table>

We can see the CPU usage increasing each time a user joins the conference, we can also notice increasing in that percentage when more motion is involved in the video transferred.

**6. Conclusion:**

In this paper, we studied the role of ICE protocol in increasing the success of WebRTC calls. And we have demonstrated the testbed implementation in our lab to test webRTC applications using different architectures in testing our applications, and we illustrated some cases showing the performance of signaling server and Turn server that could draw the basic line to write applications to measure WebRTC performance.
References


Appendices

APPENDIX1 : Test-Bed Considerations

-Connecting to Routers:
As we see in the physical architecture, we have 8 routers. To configure the routers, we need a Serial cable to connect to the Console port in the router. And use a software like ttermpro to access them.

-Hubs and routers:
each router is connected to a hub, so we have 8 hubs also, in order to connect a user agent or a client to a router, you must connect it to the corresponding HUB, please put in consideration giving the right IP while connecting user agents to the hubs.

-Naming Conventions:
Naming Convention used for the Domains is based on the rules:
1– the highest Domain in the Testbed gets one digit. For example in our Testbed, D1, D2 and D3.
2- the domain that is a subdomain(lower level) from a higher level domain, takes the previous domain digits and adds one digit to it, for example the second layer domains D11, D21.
3 – any devices connected to These domains, will add a letter after the domain name to be distinguished from domain name, for example, D21A is a computer connected to domain D21.

-Passwords:
The password of the 8 router has the form : VoIP_RouterX where X is the router name in capital, for example: routerA : VoIP_RouterA.
For the user agents and servers, the password would be Y3sW!res.

-Stun Server:
The Stun server could be separately installed on a computer that is connected to router A (user agent 1). The Stunserver I used alternatively is called Minisips that works on windows, it binds the IP of the computer with port 3478 and act as a Stun server. It is turned off by default. The stunsrver used is the one included with the turnserver in the same application.
**-Turn server:**
The Turn server is installed on a separate Machine with asterisk on in linux environment (Centos – 6.4). This machine were connected to the testbed in two scenarios, the first one is on Domain D1, the IP would be 64.131.109.19, the second would be on domain D11, The IP would be 192.168.0.5

The turn server software is available from many sources. I found out that the best two were from source forge: http://turnserver.sourceforge.net/ and the other one would be https://code.google.com/p/rfc5766-turn-server/ which is the RFC5766 Turn server. The second one has more documents and cases implemented so it would be better to install it to get more support.

Both versions has a configuration file that should be configured correctly in order to make TurnServer work. The default path for this file is:
 `/etc/turnserver/turnserver.conf`

You can run the turnserver in the command line in Linux with the command:
`turnserver -c /etc/turnserver/turnserver.conf`

The turn server will be listening on port 3478 for UDP.

**APPENDIX2 : Router Configurations**

Router A

```bash
!
version 12.4
service timestamps debug datetime msec
service timestamps log datetime msec
no service password - encryption
!
hostname RouterA
!
boot - start - marker
boot - end - marker
!
enable password VoIP_RouterA
!
no aaa new - model
!
resource policy
!
no network - clock - participate slot 1
no network - clock - participate wic 0
ip cef
!
interface FastEthernet0 /0
ip address 64.131.109.3 255.255.255.248
duplex auto
speed auto
!
interface FastEthernet0 /1
ip address 64.131.109.17 255.255.255.248
duplex auto
```
speed auto
!
router rip
network 64.0.0.0
!
ip default - gateway 64.131.109.2
ip route 0.0.0.0 0.0.0.0 64.131.109.2
!
ip http server
no ip http secure - server
no ip nat service sip udp port 5060
! control - plane
!
line con 0
line aux 0
line vty 0 4
login
!
end

Router B
!
version 12.3
service timestamps debug datetime msec
service timestamps log datetime msec
no service password - encryption
!
hostname RouterB
!
boot - start - marker
boot - end - marker
!
enable password VoIP_RouterB
!
no network - clock - participate slot 1
no network - clock - participate wic 0
no aaa new - model
ip subnet - zero
ip cef
!
interface FastEthernet0 /0
ip address 64.131.109.1 255.255.255.248
duplex auto
speed auto
!
interface FastEthernet0 /1
ip address 64.131.109.33 255.255.255.248
duplex auto
speed auto
!
router rip
network 64.0.0.0
!
no ip http server
ip classless
ip route 0.0.0.0 0.0.0.0 64.131.109.2
!
line con 0
line aux 0
line vty 0 4
login
!
end

Router C

!
version 12.1
no service single - slot - reload - enable
service timestamps debug uptime
service timestamps log uptime
no service password - encryption
!
hostname RouterC
!
logging rate - limit console 10 except errors
enable password VoIP_RouterC
!
no ip finger
!
interface FastEthernet0 /0
ip address 64.131.109.6 255.255.255.248
duplex auto
speed auto
!
interface FastEthernet0 /1
ip address 64.131.109.41 255.255.255.248
duplex auto
speed auto
!
routing rip
network 64.0.0.0
!
ip classless
ip route 0.0.0.0 0.0.0.0 64.131.109.2
no ip http server
!
line con 0
transport input none
line aux 0
line vty 0 4
login
!
end

Router D

!
version 12.3
service timestamps debug datetime msec
service timestamps log datetime msec
no service password - encryption
!
hostname RouterD
!
boot - start - marker
boot - end - marker
!
enable secret 5 $1$3NMA$KWzp / YXCKzC5Nch0mzOYf0
!
no aaa new - model
ip subnet - zero
ip cef
!
ip audit po max - events 100
!
interface Ethernet0 /0
ip address 64.131.109.42 255.255.255.248
ip nat outside
half - duplex
!
interface Serial0 /0
no ip address
shutdown
!
interface Ethernet0 /1
ip address 10.10.10.1 255.255.0.0
ip nat inside
half - duplex
!
router rip
network 64.0.0.0
!
ip nat pool publicip 64.131.109.42 64.131.109.42 prefix - length 29
ip nat inside source list 1 pool publicip overload
no ip http server
no ip http secure - server
ip classless
ip route 0.0.0.0 0.0.0.0 64.131.109.2
!
access - list 1 permit 10.10.0.0 0.0.255.255
!
line con 0
line aux 0
line vty 0 4
!
end

Router E
!
version 12.3
service timestamps debug datetime msec
service timestamps log datetime msec
no service password - encryption
!
hostname RouterE
!
boot - start - marker
boot system flash c2691 - ik9o3s - mz .123 _10 . bin
boot - end - marker
!
enable password VoIP_RouterE
!
no network - clock - participate slot 1
no aaa new - model
22ip subnet - zero
!
ip cef
!
ip audit po max - events 100
no ftp - server write - enable
!
interface FastEthernet0 /0
ip address 64.131.109.34 255.255.255.248
ip access - group 101 in
ip access - group 101 out
ip nat outside
duplex auto
speed auto
!
interface FastEthernet0 /1
ip address 192.168.1.1 255.255.255.0
ip access - group 101 in
ip access - group 101 out
ip nat inside
duplex auto
speed auto
!
interface FastEthernet1 /0
no ip address
shutdown
duplex auto
speed auto
!
interface FastEthernet1 /1
no ip address
shutdown
duplex auto
speed auto
!
router rip
network 64.0.0.0
!
ip nat pool publicip 64.131.109.34 64.131.109.34 prefix - length 29
ip nat inside source list 1 pool publicip overload
ip nat inside source static udp 192.168.1.5 3478 64.131.109.34
3478 extendable
ip nat outside source static udp 192.168.1.5 3478 64.131.109.34
3478 extendable
no ip http server
no ip http secure - server
ip classless
ip route 0.0.0.0 0.0.0.0 64.131.109.2 !
access - list 1 permit 192.168.1.0 0.0.0.255
access - list 101 permit ip any any !
line con 0
transport preferred all
transport output all
line aux 0
23transport preferred all
transport output all
line vty 0 4
login
transport preferred all
transport input all
transport output all !
end

Router F !
version 12.4
service timestamps debug datetime msec
service timestamps log datetime msec
no service password - encryption
!
hostname RouterF !
boot - start - marker
boot - end - marker !
enable password VoIP_RouterF !
no aaa new - model
no network - clock - participate slot 1
no network - clock - participate wic 0
ip cef !
ip auth - proxy max - nodata - conns 3
ip admission max - nodata - conns 3 !
interface FastEthernet0 /0
ip address 192.168.1.2 255.255.255.0
ip access - group 102 in
ip nat outside
ip virtual - reassembly
duplex auto
speed auto !
interface FastEthernet0 /1
ip address 172.16.2.1 255.255.255.0
ip nat inside
ip virtual - reassembly
duplex auto
speed auto
!
ip forward - protocol nd
ip route 0.0.0.0 0.0.0.0 192.168.1.1
!
no ip http server
no ip http secure - server
ip nat pool publicip 192.168.1.2 192.168.1.2 prefix - length 24
ip nat inside source list 1 pool publicip overload 24!
access - list 1 permit 172.16.2.0 0.0.0.255
access - list 102 deny ip 64.131.109.40 0.0.0.7 any
access - list 102 permit ip any any
!
control - plane
!
voice - port 1/0/0
!
voice - port 1/0/1
!
gatekeeper
shutdown
!
line con 0
line aux 0
line vty 0 4
login
!
end

Router G
!
version 12.2
service timestamps debug uptime
service timestamps log uptime
no service password - encryption
!
hostname RouterG
!
enable password VoIP_RouterG
!
ip subnet - zero
!
interface FastEthernet0 /0
ip address 10.10.10.2 255.255.0.0
ip access - group 102 in
ip nat outside
duplex auto
speed auto
!
interface FastEthernet0 /1
ip address 172.16.3.1 255.255.255.0
ip nat inside
duplex auto
speed auto
! interface Hssi1 /0
no ip address
shutdown
!
ip nat pool publicip 10.10.10.2 10.10.10.2 prefix - length 16
ip nat inside source list 1 pool publicip overload
ip classless
25ip route 0.0.0.0 0.0.0.0 10.10.10.1
no ip http server
ip pim bidir - enable
!
access - list 1 permit 172.16.3.0 0.0.0.255
access - list 102 deny ip 64.131.109.32 0.0.0.7 any
access - list 102 permit ip any any
!
line con 0
line aux 0
line vty 0 4
login
!
end

Router H
!
version 12.2
service timestamps debug datetime msec
service timestamps log datetime msec
no service password - encryption
!
hostname RouterH
!
logging queue - limit 100
enable secret 5 $1$el$CAYn9UFVLlIy7nAJhKKACd1
!
memory - size iomem 15
ip subnet - zero
!
no voice hpi capture buffer
no voice hpi capture destination
!
mta receive maximum - recipients 0
!
interface FastEthernet0 /0
ip address 64.131.109.4 255.255.255.248
ip nat outside
duplex auto
speed auto
!
interface FastEthernet0 /1
ip address 192.168.0.1 255.255.255.0
ip access - group 102 out
ip nat inside
duplex auto
speed auto
!
interface FastEthernet1 /0
no ip address
shutdown
duplex auto
speed auto
!
interface FastEthernet1/1
no ip address
shutdown
duplex auto
speed auto
!
router rip
network 64.0.0.0
!
ip nat pool publicip 64.131.109.4 64.131.109.4 prefix length 29
ip nat inside source list 1 pool publicip overload
ip nat inside source static udp 192.168.0.5 3478 64.131.109.4 3478 extendable
ip nat inside source static udp 192.168.0.5 5060 64.131.109.4 5060 extendable
no ip http server
no ip http secure - server
ip classless
ip route 0.0.0.0 0.0.0.0 64.131.109.2
!
access - list 1 permit 192.168.0.0 0.0.0.255
access - list 102 permit udp any any eq 3478
access - list 102 permit udp any any eq 5060
access - list 102 deny ip any any
!
call rsvp - sync
!
mgcp profile default
!
dial - peer cor custom
!
line con 0
line aux 0
line vty 0 4
login
!
end
APPENDIX3 : Simple Audio/Video Chat software code :

The HTML Page Code:

```html
<!DOCTYPE html>
<!-- saved from url=(0027)http://64.131.109.21/Elias/ -->
<html><head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>IIT WebRTC Demo</title>
<script src="./IIT WebRTC Demo_files/adapter.js"></script>
<link href="./IIT WebRTC Demo_files/style.css" type="text/css" rel="stylesheet" media="all">
<link rel="shortcut icon" href="http://64.131.109.21/img/favicon.ico" type="image/x-icon">
</head>
<body>

<div class="container">
  <div class="logo">
    <p class="top_heading">WebRTC Project</p>
    <img class="logo_img" src="./IIT WebRTC Demo_files/cpd_logo.gif" alt="IIT ITM Logo">
  </div>
</div>

<br>
<button id="btn1" onclick="start()">Start</button>
```

<div class="bottom_banner">
  <p class="name">Student, Elias Yousef. Director, Professor Carol Davids Real-Time Communications Laboratory, Illinois Institute of Technology, IIT School of Applied Technology</p>
  <p class="address">3434 S State St    Chicago, IL  60616</p>
</div>

<script src="./IIT WebRTC Demo_files/socket.io.js"></script>
<script>
var socket = io.connect('http://64.131.109.20:8080/');
//64.131.109.20
var pc;

// call start() to initiate
function start() {
  //var configuration =null;
  //var configuration
  ={"iceServers":["stun:64.131.109.20:3478"]}; //STUN Local
  var configuration ={"iceServers":
  ["turn:Elias@64.131.109.4:3478", "credential":"Elias1986#1"]}; //TURN

  pc = new RTCPeerConnection(configuration);
  console.log('Peer Connection created');
  // send any ice candidates to the other peer
  pc.onicecandidate = function (evt) {
    if (evt.candidate){
      var cand={ "candidate": evt.candidate };
      socket.send(JSON.stringify(cand));
      console.log("ICE envoye")
    }
  };

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = function () {
    pc.createOffer(localDescCreated, logError);
  }

  // once remote stream arrives, show it in the remote video element
  pc.onaddstream = function (evt) {
    remoteView.src = URL.createObjectURL(evt.stream);
  };

  // get a local stream, show it in a self-view and add it to be sent
  getUserMedia({audio: true, video: true }, function (stream) {
    selfView.src = URL.createObjectURL(stream);
    pc.addStream(stream);
  });
</script>
function localDescCreated(desc) {
    pc.setLocalDescription(desc, function () {
        socket.send(JSON.stringify({'sdp': pc.localDescription}));
        console.log("description sdp sent");
    }, logError);
}

//Handle the interaction webserver -> browser
socket.on('message', onMessage);
function onMessage(evt) {
    //If the browser doest not have a PC, it is the moment...
    if (!pc)
        start();
    var message = JSON.parse(evt);
    console.log(message);
    if (message.sdp){
        console.log('sdp received');
        pc.setRemoteDescription(new RTCSessionDescription(message.sdp),
        function () {
            // if we received an offer, we need to answer
            if (pc.remoteDescription.type == "offer")
                pc.createAnswer(localDescCreated, logError);
        }, logError);
    } else{
        pc.addIceCandidate(new RTCIceCandidate(message.candidate));
        console.log('ICE recu');
    }
}

function logError(error) {
    log(error.name + ': ' + error.message);
}

function stop(){
    pc.close();
}
</script>

</div></body></html>

The Srver.JS code:
// create the http server and listen on port
var server = require('http').createServer();
var app = server.listen(8080, function() {
    console.log((new Date()) + " Server is listening on port 8080");
});

// create the socket server on the port
var io = require('socket.io').listen(app);
```javascript
var sockets = require('json-sockets');

// This callback function is called every time a socket tries to connect to the server
io.sockets.on('connection', function(socket) {

    console.log((new Date()) + ' Connection established.);

    // When a user send a SDP message
    // broadcast to all users in the room
    socket.on('message', function(message) {
        console.log((new Date()) + ' Received Message, broadcasting: ');
        socket.broadcast.emit('message', message);
    });

    // When the user hangs up
    // broadcast bye signal to all users in the room
    socket.on('disconnect', function() {
        // close user connection
        console.log((new Date()) + ' Peer disconnected.);
        socket.broadcast.emit('user disconnected');
    });
});

APPENDIX4 : Performance Test Socket.IO applications :

Server Javascript Code:

// var profiler = require ( ' v8 - profiler ' ) ;
var io = require ( ' socket . io ' ) . listen (3000) ;
var exec = require ( ' child_process ' ) . exec ;
io . configure ( function () {
    io . set ( ' log level ' , 1) ;
    if ( transport ) {
        io . set ( ' transports ' , [ transport ]);}
    }
    });
// command to read process consumed memory and cpu time
var get_cpu_command = " ps -p " + process . pid + " -u | grep " + process . pid ;
var users = 0;
var countReceived = 0;
var countSent = 0;
var tab_users = {};
var i = 0;
var i_tab = 0;
var sum = 0;
function roundNumber ( num , precision ) {
    return parseFloat ( Math . round ( num * Math . pow (10 , precision ) ) ) / Math . pow (10 , precision ) ;
}
```

davids@iit.edu  Project Report  33
setInterval ( function () {
    var auxReceived = roundNumber ( countReceived / users , 1)
    var msuReceived = ( users > 0 ? auxReceived : 0) ;
    var auxsended = roundNumber ( countSended / users , 1)
    var msuSended = ( users > 0 ? auxSended : 0) ;
    // call a system command ( ps ) to get current process resources
    utilization
    var child = exec ( getCpuCommand , function ( error , stdout , stderr )
    {
        var s = stdout . split (/
        s +/) ;
        var cpu = s [2];
        var memory = s [3];
        var l = [ 'U : ' + users + '; ' ,
                 ' MR / S : ' + countReceived + '; ' ,
                 ' MS / S : ' + countSended + '; ' ,
                 ' CPU : ' + cpu + '; ' ,
                 ' nb Transactions : ' + sum + '; ' ,
                 ' Mem : ' + memory + '; ' ];
        /*
        var l = [ countReceived + '; ' + countSended + '; ' + cpu + '; ' + memory + 
                 '; ' ]; */
        // sum = sum + countReceived ;
        console . log ( l . join ( ' ,
        t ') ) ;
        countReceived = 0;
        countSended = 0;
    });
}, 100) ;
io . sockets . on ( ' connection ' , function ( socket ) {
    tab_users [ i_tab ]= socket . id ; // record each client in the array
    i_tab = i_tab +1;
    socket . on ( ' message ' , function ( message ) {
        users ++;
        countReceived ++;
        for ( i =0; i < i_tab ; i ++) {
            if ( tab_users [ i ]== socket . id ) {
                socket . send ( message ) ;
                i = i_tab ; // STOP LOOP
                countSended ++;
            }
        }
    };
    socket . on ( ' disconnect ' , function () {
        for ( i =0; i < i_tab ; i ++) {
            if ( tab_users [ i ]== socket . id ) {
                delete ( tab_users [ i ]);
                sum = sum +1;
            }
        }
        users --;
        i_tab --;
    })
}
Client Javascript:

```javascript
var io = require ( 'socket.io-client' );
var message = "SDP message, this variable is modified so it can fit in the Appendices";
function user ( host , port ) {
    var socket = io . connect ( 'http://' + host + ':' + port , { 'force new connection ': true }) ;
    var stop =0;
    socket . on ( 'connect' , function () {
        if ( stop ==0) {
            // console.log('0');
            stop =1;
            socket . send ( message ) ; // SDP request
        }
    socket . on ( 'message' , function ( message ) {
        if ( stop ==1) { // SDP request received
            socket . send ( message ) ; // send SDP response
            stop =2;
        }
        if ( stop ==2) { // SDP response received
            socket . disconnect () ;
        }
    }) ;
    var id ;
    var id_t ;
    id = setInterval ( function () { user ( 'localhost' , '3000' ) ;
    }, 20) ; // one user each X ms

Multithreading C Code

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#define NBTHREADS 3
pthread_t tid [ NBTHREADS ];
void * process () {
    system ("node c_benchmark.js");
    return (0);
}
int main () {
    int i =0;
    int j =0;
    while (i < NBTHREADS ) {
        pthread_create (&( tid [ i ]) , NULL , & process , NULL );
        i ++;
    }
    while (j < NBTHREADS ) {
```
pthread_join (tid[j], NULL);
    j++;
}
return 0;
}