UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea Magistrale in
Informatica

Elaborato finale

The Wormhole Peer Sampling Service implemented over WebRTC

Supervisore

Alberto Montresor

Laureando

Davide Spadini

Anno accademico 2015/2016

Ringraziamenti

...thanks to...

Indice

Sommario

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sed nunc orci. Aliquam nec nisl vitae sapien pulvinar dictum quis non urna. Suspendisse at dui a erat aliquam vestibulum. Quisque ultrices pellentesque pellentesque. Pellentesque egestas quam sed blandit tempus. Sed congue nec risus posuere euismod. Maecenas ut lacus id mauris sagittis egestas a eu dui. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Pellentesque at ultrices tellus. Ut eu purus eget sem iaculis ultricies sed non lorem. Curabitur gravida dui eget ex vestibulum venenatis. Phasellus gravida tellus velit, non eleifend justo lobortis eget.

Sommario è un breve riassunto del lavoro svolto dove si descrive l'obiettivo, l'oggetto della tesi, le metodologie e le tecniche usate, i dati elaborati e la spiegazione delle conclusioni alle quali siete arrivati.

Il sommario dell'elaborato consiste al massimo di 3 pagine e deve contenere le seguenti informazioni:

- contesto e motivazioni
- breve riassunto del problema affrontato
- tecniche utilizzate e/o sviluppate
- risultati raggiunti, sottolineando il contributo personale del laureando/a

1 WebRTC

WebRTC is an Application Programming Interface (API) definition drafted by the World Wide Web Consortium (W3C) that supports browser-to-browser applications for voice calling, video chat, and Peer-To-Peer (P2P) file sharing without plugins. It is already implemented in the Chrome, Firefox and Opera browsers. The purpose of WebRTC is to enable rich, high-quality RTC applications to be developed for the browser, mobile platforms, and IoT devices, and allow them all to communicate via a common set of protocols.

From the point of view of an end-user, WebRTC provides a much simpler way to have real-time conversation with another end-user. It is based on browser and Internet which almost all personal or enterprise computers already have, hence without any installation and plugins the users can have exactly the same service which previous stand-alone desktop client provides. WebRTC makes these capabilities accessible to web developers via standard HTML5 tags and JavaScript APIs. For example, we can consider functionality similar to that offered by Skype[1], but without installing any software or plug-ins.

The codecs and protocols used by WebRTC do a huge amount of work to make real-time communication possible, even over unreliable networks (e.g. packet loss concealment, echo cancellation, bandwidth adaptivity, image cleaning, ...)[2]. However, WebRTC API still requires a lot of works in order to successfully create the connection between two peers. A fully explation of this process is covered in the Sect.[1.2]. However, in this project I use the EasyRTC[1] framework in order to simplify the creation of the network and it maintenance. More details about this in Sect.[1.4].

1.1 WebRTC APIs

The main tasks of WebRTC are:

- Acquiring audio and video: getting access to the microphone or camera, getting a streaming of media for either of them
- Communicating audio and video: being able to connect to another WebRTC end-point through Internet, and send audio and video stream in real-time
- Communicating generic data: not only audio and video, but for any arbitrary application data

These three main categories are translated in three Javascript APIs:

- **MediaStream (aka getUserMedia)**
- **RTCPeerConnection**
- **RTCDataChannel**

MediaStream is not relevant for this project, in fact the messages that the peers will exchange are not streams of audio and video, but simple JSON.

RTCDataChannel enables peer-to-peer exchange of arbitrary data, with low latency and high throughput. This is what I use to send data between peers, but still they need to be connected in order to send and receive messages, so the DataChannel is built on top of a *PeerConnection*.

RTCPeerConnection is the WebRTC component that handles stable and efficient communication of streaming data between peers. In Fig.1.1 the WebRTC architecture diagram shows the role of RTCPeerConnection: the main thing to understand from this diagram is that RTCPeerConnection

[1]Skype is a free voice-over-IP service and instant messaging client, currently developed by the Microsoft Skype Division.

shields web developers from the myriad complexities that lurk beneath. As explained before, I did not use WebRTC for audio and video streaming, so I do not comment the first two column of the diagram, instead in the next sections I show how to establish a connection using RTCPeerConnection.

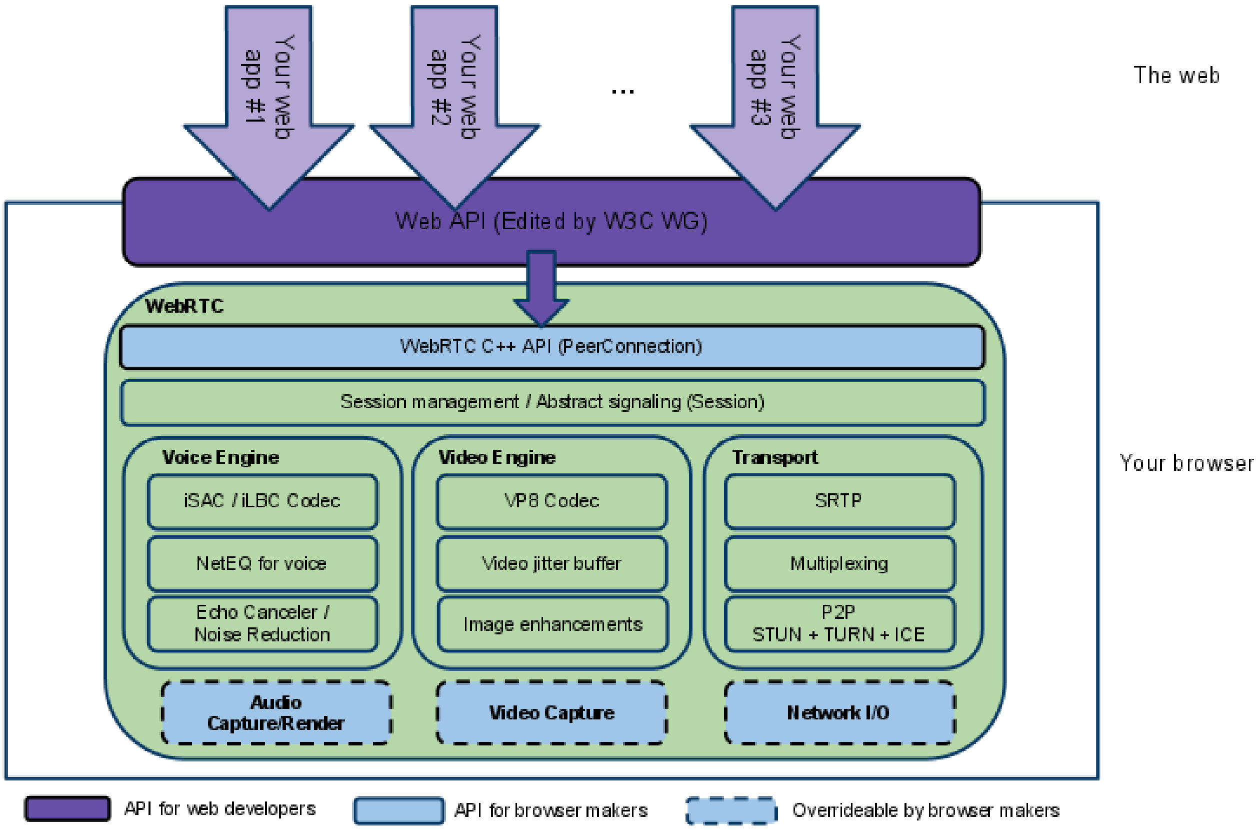

Figura 1.1: WebRTC architecture (from `webrtc.org`)

1.2 WebRTC: Signaling

WebRTC uses RTCPeerConnection to create a connection between peers and communicate audio and video. In order to establish the connection between them it needs a mechanism to coordinate the communication and to send control messages, a process known as signaling.

Signaling is used to initialize the connection and exchange three types of information:

- **Session control messages**: to initialize or close communication and report errors.
- **Network configuration**: to the outside world, what's my computer's IP address and port?
- **Media capabilities**: what codecs and resolutions can be handled by my browser and the browser it wants to communicate with?

The exchange of information via signaling must have completed successfully before peer-to-peer streaming can begin.

Signaling methods and protocols are not specified by WebRTC: signaling is not part of the RTC-PeerConnection API. So the web developer can choose the messaging protocol he prefer, for example in my case I use WebSockets. The reason why the WebRTC group has made this decision is to avoid redundancy and to maximize compatibility with established technologies [5].

Let see an example of how to use RTCPeerConnection: imagine Alice wants to communicate with Bob. To initialize this process RTCPeerConnection has two tasks:

- Ascertain local media conditions, such as resolution and codec capabilities.
- Get potential network addresses for the application's host, known as candidates. (see Sect.[1.3])

For the first point, the exchange of media configuration information proceeds using an offer/answer mechanism that is called JSEP, JavaScript Session Establishment Protocol [5]. Fig.1.2 shows the JSEP architecture: both the caller and callee have to save their local session description taken from the browser and send them through some signaling mechanism, then when they receive the session description of the other they set it as the remote session description. Once the process is finished, they both know the configuration of the peer they want to communicate with.

The entire sequence of steps is the following:

- Alice creates an RTCPeerConnection object.
- Alice creates an offer using the RTCPeerConnection *createOffer()* method.
- Alice set her local description to her offer.
- Alice uses a signaling mechanism to send her offer to Bob.
- Bob set his remote description to Alice's offer, so that his RTCPeerConnection knows about Alice's setup.
- Bob create an answer using the *createAnswer()* function.
- Bob sets his answer as the local description.
- Bob then uses the signaling mechanism to send his answer back to Alice.
- Alice sets Bob's answer as the remote session description.

Offers and answers are communicated in Session Description Protocol format (SDP) [4], which look like this:

```
v=0
o=- 7614219274584779017 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE audio video
a=msid-semantic: WMS
m=audio 1 RTP/SAVPF 111 103 104 0 8 107 106 105 13 126
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:W2TGCZw2NZHuwlnf
a=ice-pwd:xdQEccP40E+P0L5qTyzDgfmW
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=mid:audio
a=rtcp-mux
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:9c1AHz27dZ9xPI91YNfSlI67/EMkjHHIHORiClQe
a=rtpmap:111 opus/48000/2
....
```

Using this format, in the offer and answer messages there are all the necessary information to guarantee that the peers can communicate using the same codecs, resolution and other media capabilities. Once this process is finished, and they both know the configuration of the other, they use the ICE Framework in order to establish the connection.

1.3 WebRTC: ICE Framework

For metadata signaling, WebRTC applications use an intermediary server, the signaling server, but for actual media and data streaming once a session is established, RTCPeerConnection attempts to connect clients directly: peer to peer.

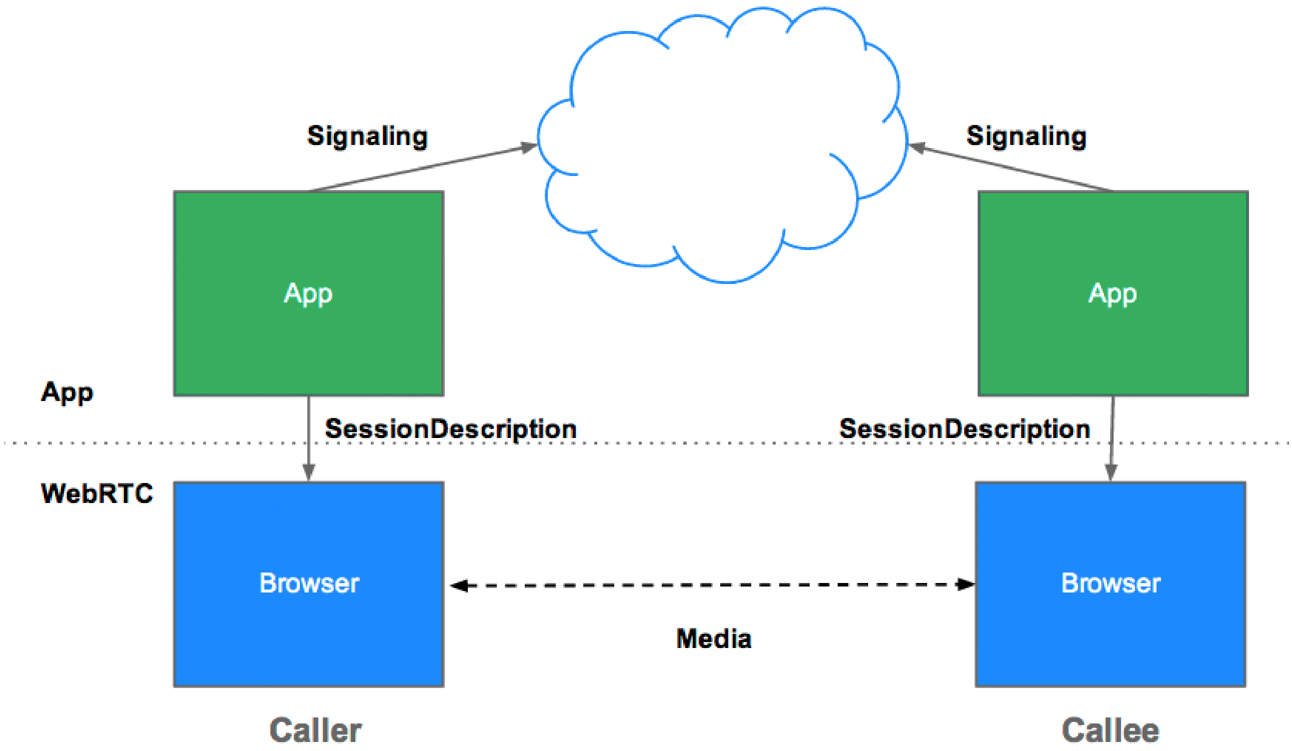

Figura 1.2: Signaling Diagram

In a perfect world, all the nodes are public and they are always reachable. In reality, this is not the case: in fact most devices are behind one or more layers of Network Address Translation (NAT)[6], some have anti-virus software that blocks certain ports and protocols, and many are behind proxies and corporate firewalls. All these configurations make the connection peer to peer impossible. However, WebRTC applications can use the *Interactive Connectivity Establishment* (ICE) framework to overcome the complexities of real-world networking.

ICE tries to find the best path to connect peers. It tries all possibilities in parallel and chooses the most efficient option that works. ICE first tries to make a connection using the host address obtained from a device's operating system and network card; if that fails (which it will for devices behind NATs) ICE obtains an external address using a *Session Traversal Utilities for NAT* (STUN) server, and if that fails, traffic is routed via a *Traversal Using Relays around NAT* (TURN) server[3].

In other words, if the direct link fails (so if the peers are behind a NAT), ICE uses the STUN server. Fig.1.3 shows how it works: the server has one simple task, find the public IP address and port of the peer and send that address back as a response. This process enables a WebRTC peer to get a publicly accessible address for itself, and then pass that to the other peer via a signaling mechanism, in order to set up a direct link.

If that fails, TURN servers can be used as a fallback. These servers have a conceptually simple task, to relay a stream, but unlike STUN servers, they inherently consume a lot of bandwidth. This is in fact the last chance of the ICE Framework.

Fig.1.4 represents the complete schema.

The URLs of the STUN and/or TURN servers are (optionally) specified by the WebRTC application in the configuration object that is the first argument of the RTCPeerConnection constructor. Once RTCPeerConnection has these information, it uses the ICE framework to work out the best path between peers, working with STUN and TURN servers as necessary. When it find the best solution, it initialize the connection and the peers can start to communicate.

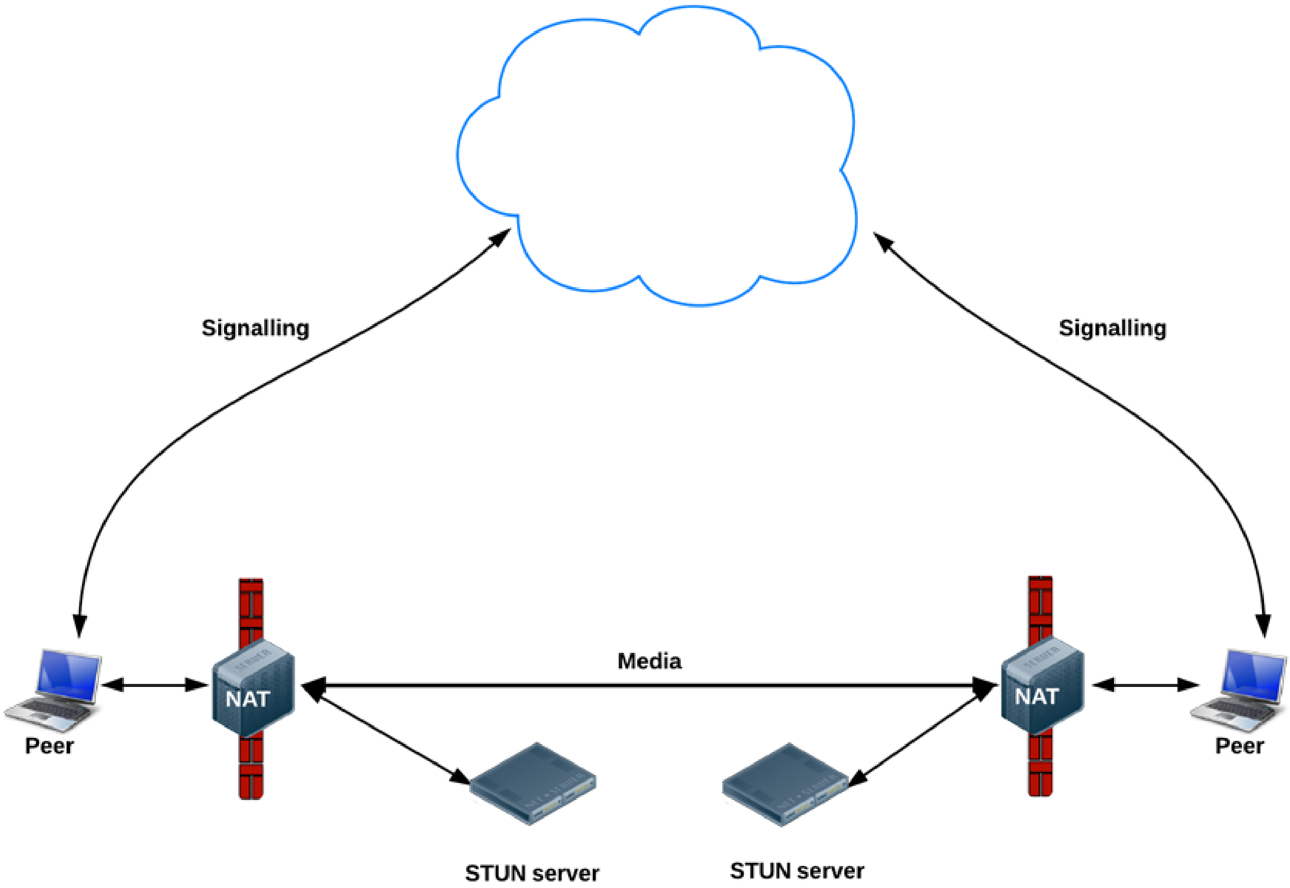

Figura 1.3: Using STUN servers to get public IP:port addresses

1.4 Back to reality: EasyRTC Framework

As shown in the previous sections, establish a connection between two peers in WebRTC is not so simple. As is often the case with software, with power comes complexity. WebRTC has a learning curve that is likely to hamper its use by web developers. To hide that complexity, Priologic[2] has built the EasyRTC framework.

One of the most important feature of this framework is that, while the WebRTC API requires the developers to implement an involved message passing scheme between clients to establish the peer to peer connection, it already provides this schema[1] and is completely hidden from the point of view of the web developers. It is really well documented and there already exists the EasyRTC server (the signaling server) that the nodes will contact in order to register themselves in the network.

EasyRTC is implemented in *Node.js* and it is available for free.

1.4.1 EasyRTC Server

The main purpose of the server is to serve the requests of "join the network" from the nodes, so it is always listening on a specific port and all the nodes have to be able to reach it. Once a node is registered in the network, has an unique identifier that will be valid until it will disconnect. This ID will identify the node within the network, so the other nodes will have to use it in order to contact that specific peer.

When a node wants to connect to another one, the request pass through this server that will take care of create the connection between them. Once the connection has been created, the nodes can communicate without passing through the server.

As explained before, the signaling mechanism is not part of the WebRTC: easyRTC server by default uses WebSockets[3]. One advantage of this technique is that through the heartbeats (already implemented inside WebSockets) the server knows which nodes are on-line and which one have di-

[2]It is a team of Canadian software Developers. More information: `https://www.priologic.com`

[3]EasyRTC server by default uses the *socket.io* module. More information: `http://socket.io`

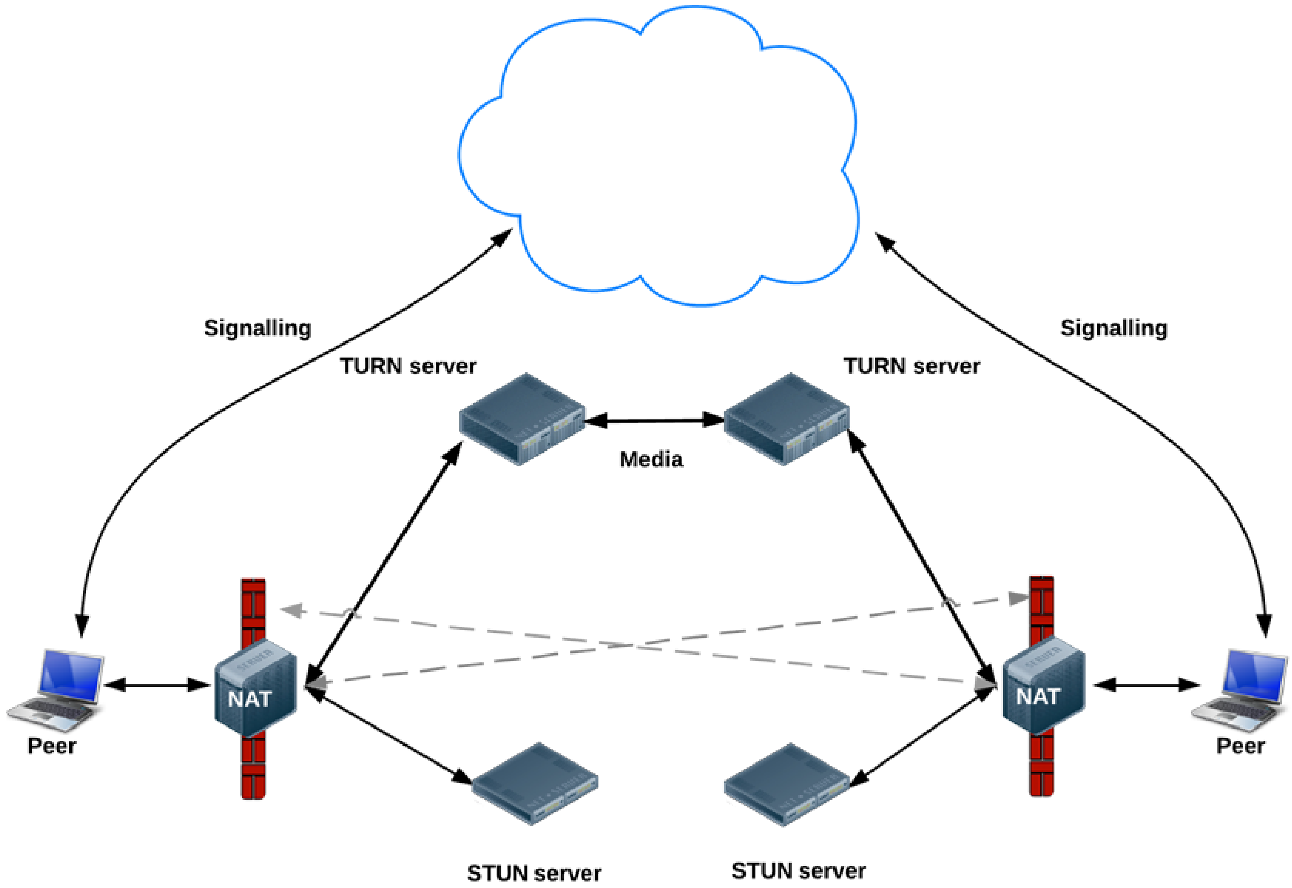

Figura 1.4: The full schema: STUN, TURN and signaling

sconnected or crashed. In fact every "N" seconds (by default N = 25 seconds), the server sends an heartbeat to the node and if the nodes does not reply within the heartbeat timeout (by default it is 60 seconds) it closes the connection with it. I uses this technique to capture the failure of a node (it could be crashed or simply disconnected) and send to all the other node an advice. If a node was connected to the failed node, it has to replace the broken link with a new one and to do so it contacts the tracker (more information in Sect.2).

1.5 SPOF

The last thought that I want to consider is about the "Single Point Of Failure" (SPOF): in my system, if the EasyRTC server crashes or for some reason it is no more reachable from the outside world, no-one can join the network and, especially, no-one can capture the failure of another node (in fact this event is triggered only by the server). Handle this situation is not part of my project, in fact one assumption is that the server is always on-line.

However, there is the possibility to add redundancy in the system, creating more than one servers in order to have some of them that acts as backup servers. Then, each node decides which of them to contact, and if for some reason that server goes down, it connects to another one. In this way the problem is solved, but as explained before this was not the task of the project, so for simplicity I implement only one server and I assume that is always reachable.

2 Tracker

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sed nunc orci. Aliquam nec nisl vitae sapien pulvinar dictum quis non urna. Suspendisse at dui a erat aliquam vestibulum. Quisque ultrices pellentesque pellentesque. Pellentesque egestas quam sed blandit tempus. Sed congue nec risus posuere euismod. Maecenas ut lacus id mauris sagittis egestas a eu dui. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Pellentesque at ultrices tellus. Ut eu purus eget sem iaculis ultricies sed non lorem. Curabitur gravida dui eget ex vestibulum venenatis. Phasellus gravida tellus velit, non eleifend justo lobortis eget.

3 Wormhole

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sed nunc orci. Aliquam nec nisl vitae sapien pulvinar dictum quis non urna. Suspendisse at dui a erat aliquam vestibulum. Quisque ultrices pellentesque pellentesque. Pellentesque egestas quam sed blandit tempus. Sed congue nec risus posuere euismod. Maecenas ut lacus id mauris sagittis egestas a eu dui. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Pellentesque at ultrices tellus. Ut eu purus eget sem iaculis ultricies sed non lorem. Curabitur gravida dui eget ex vestibulum venenatis. Phasellus gravida tellus velit, non eleifend justo lobortis eget.

4 Evaluation

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sed nunc orci. Aliquam nec nisl vitae sapien pulvinar dictum quis non urna. Suspendisse at dui a erat aliquam vestibulum. Quisque ultrices pellentesque pellentesque. Pellentesque egestas quam sed blandit tempus. Sed congue nec risus posuere euismod. Maecenas ut lacus id mauris sagittis egestas a eu dui. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Pellentesque at ultrices tellus. Ut eu purus eget sem iaculis ultricies sed non lorem. Curabitur gravida dui eget ex vestibulum venenatis. Phasellus gravida tellus velit, non eleifend justo lobortis eget.

Bibliografia

[1] Easyrtc. https://easyrtc.com/docs/guides/.

[2] Getting started with webrtc. http://www.html5rocks.com/en/tutorials/webrtc/basics/.

[3] Webrtc in the real world: Stun, turn and signaling. http://www.html5rocks.com/en/tutorials/webrtc/infrastructu

[4] Google. Sdp for the webrtc. *Internet-Draft*, August 04, 2015.

[5] Google. Javascript session establishment protocol. *Internet-Draft*, February 25, 2013.

[6] P. Srisuresh and M. Holdrege. *ip network address translator (nat) terminology and considerations.* Rfc-2663, 1999.