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 plugins.

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 work 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 we use the EasyRTC[1] framework in order to simplify the creation of the network and its 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 we 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, we did not use WebRTC for audio and video streaming (the first two columns of the diagram); instead, the next sections focus on how to establish a connection using RTCPeerConnection.

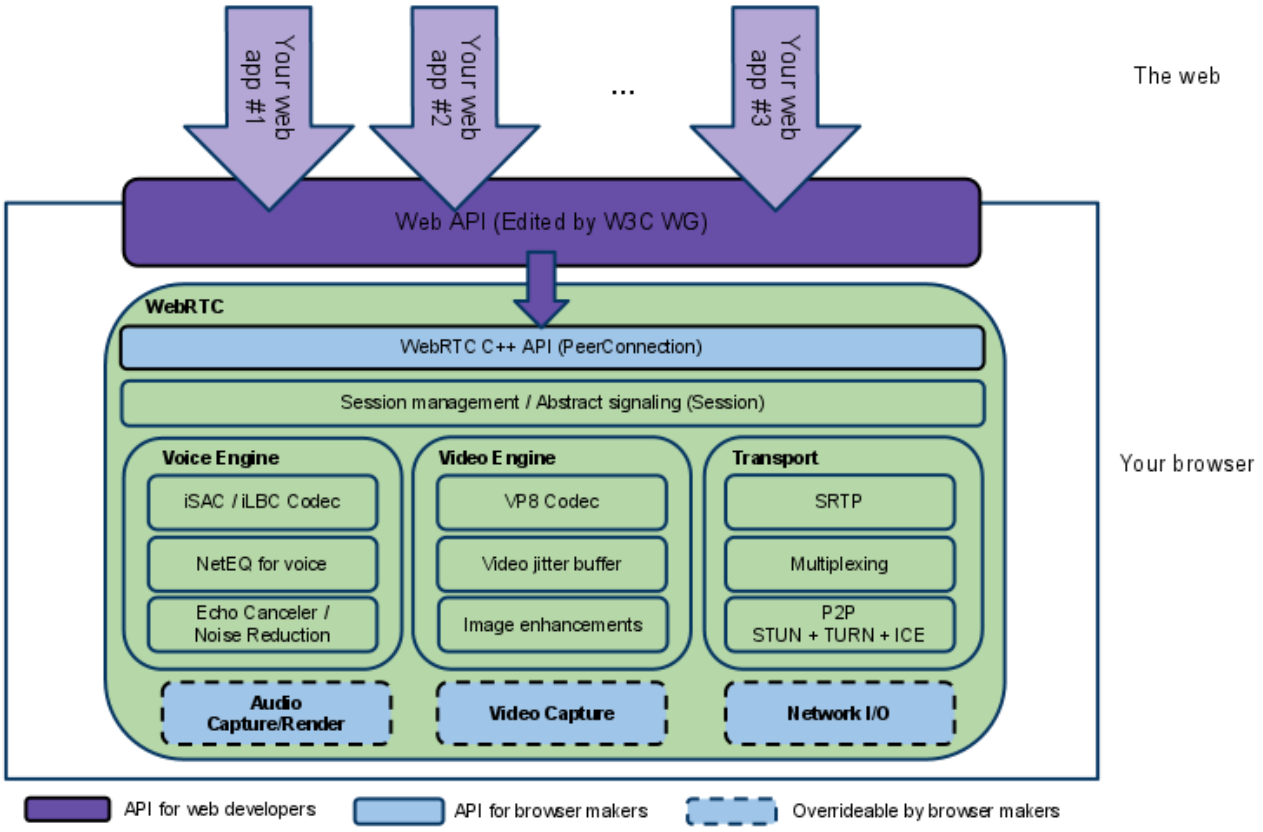

Figura 1.1: WebRTC architecture (from `webrtc.org`)

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 is 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 successfully completed before peer-to-peer streaming can begin.

Signaling methods and protocols are not specified by WebRTC: signaling is not part of the RTCPeerConnection API. So the web developer can choose the messaging protocol he/she prefer, for example in my case we use WebSockets. The reason why the WebRTC group has made this decision is to avoid redundancy and to maximize compatibility with established technologies [9].

Let us 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 [9]. 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 sets her local description to her offer.
- Alice uses a signaling mechanism to send her offer to Bob.
- Bob sets 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) [8], 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.

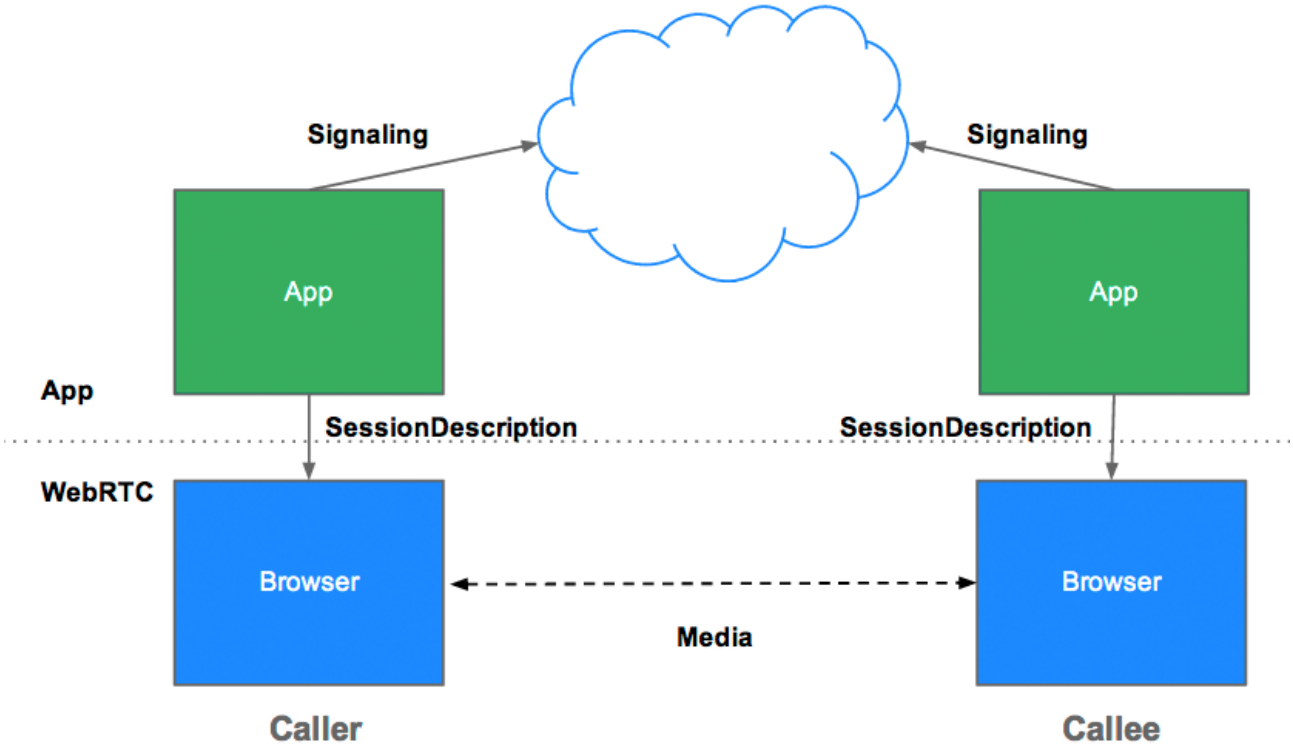

Figura 1.2: Signaling Diagram

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)[14], 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.

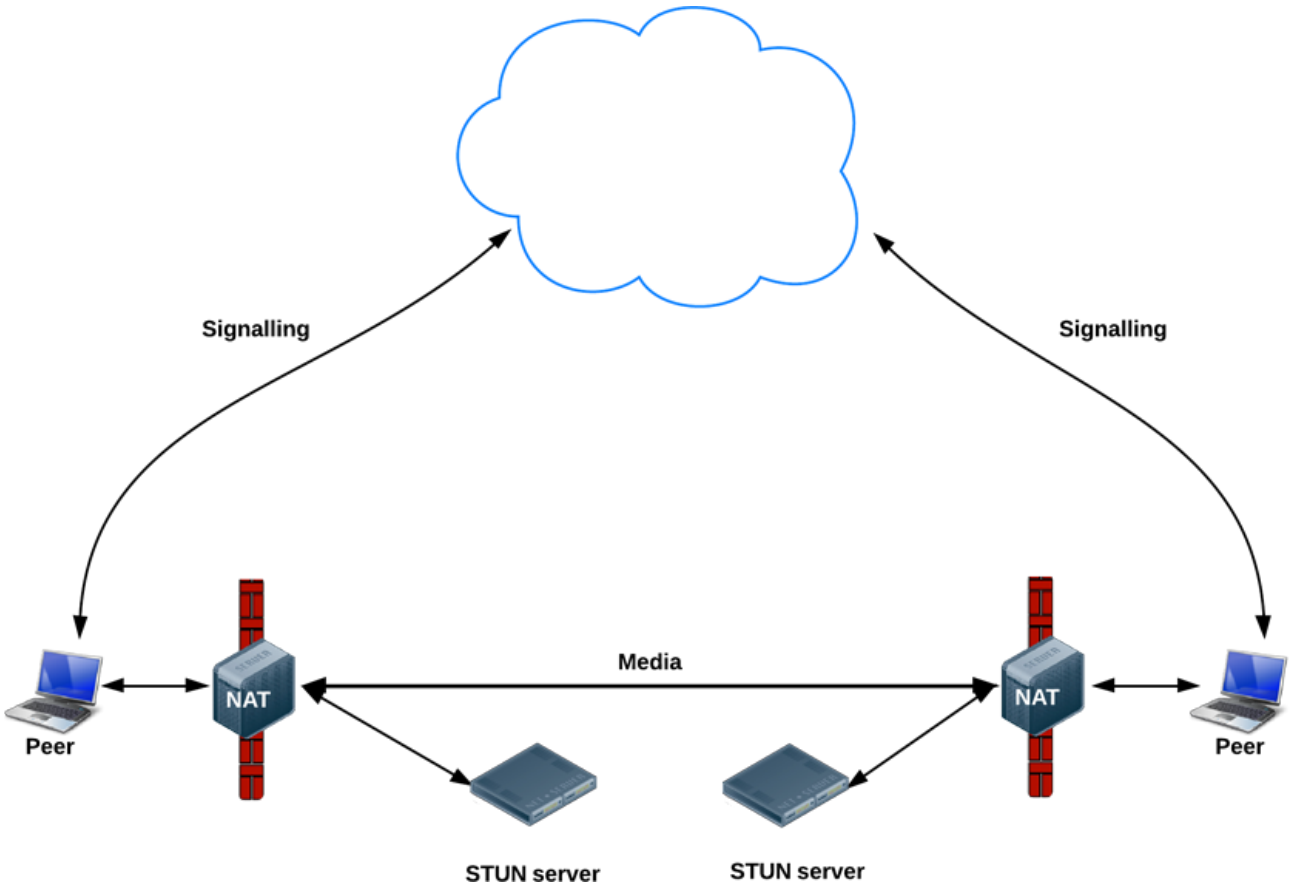

Figura 1.3: Using STUN servers to get public IP:port addresses

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 discon-

[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`

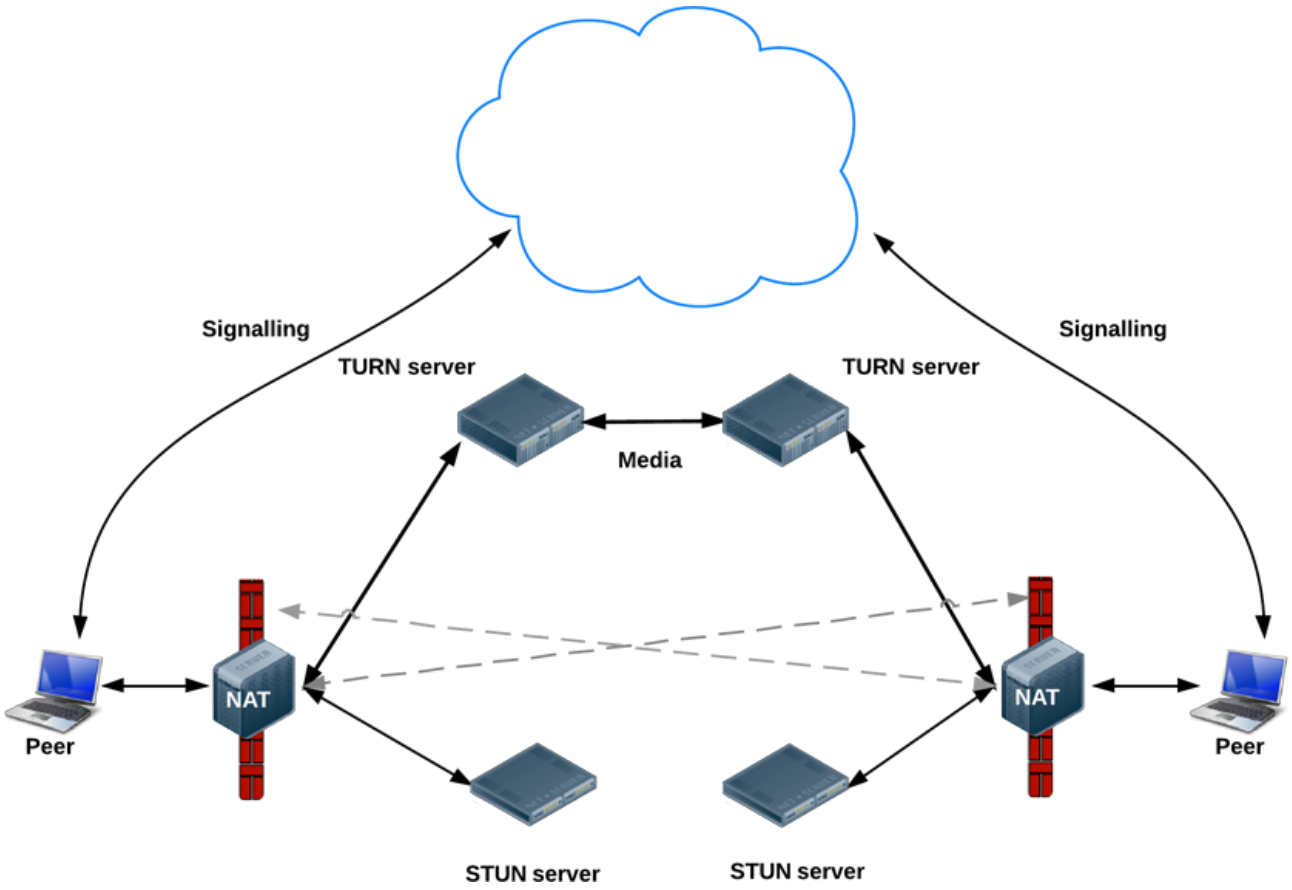

Figura 1.4: The full schema: STUN, TURN and signaling

nected 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. We use 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

A final issue to be considered is related to 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 we implement only one server and we assume that is always reachable.

2 Tracker

In the previous sections we saw how a node can initialize a connection to another node using EasyRTC Framework: at the beginning they both have to contact the EasyRTC Server to obtain the ID, and then pass the identifier of the other node to the *connect* function in order to establish the connection. However, EasyRTC does not specify how the nodes have to exchange these identifiers: for this purpose we created a tracker.

When the nodes bootstrap, they have to contact the EasyRTC Server in order to join the network and then they have to register themselves to the tracker. In this way the latter will eventually have the complete list of the nodes present in the network. When a peer needs to connect to another one (i.e. to replace a broken link):

- it asks the tracker a reference of a new node
- the tracker chooses an ID at random from the list and it returns it to the peer
- the peer analyzes the answer and connects to that ID

We use the tracker to provide all nodes with a list of "initial peers" to contact in order to start the algorithm (more information in Sect.3), to replace the broken links and for all the necessary replacements expected from the algorithm. The main tasks of the tracker are shown in Fig. 2.1.

In this project the tracker acts like a central server: we only have one tracker for all the nodes, and it has to handle and serve all the requests that derive from the nodes. However, this is not a problem since it has relatively low load (the link replacements are rare) and the algorithm is designed to reduce the number of new network connections created.

The tracker is hosted on Heroku[1] and is implemented in *Node.js*. The nodes contact the tracker using XMLHttpRequest (XHR)[2] and then they parse the response to obtain what they asked.

[1]Heroku is a cloud platform that lets companies build, deliver, monitor and scale apps `https://www.heroku.com/`

[2]XMLHttpRequest is an API available to web browser scripting languages such as JavaScript that is used to send HTTP or HTTPS requests to a web server and load the server response data back into the script

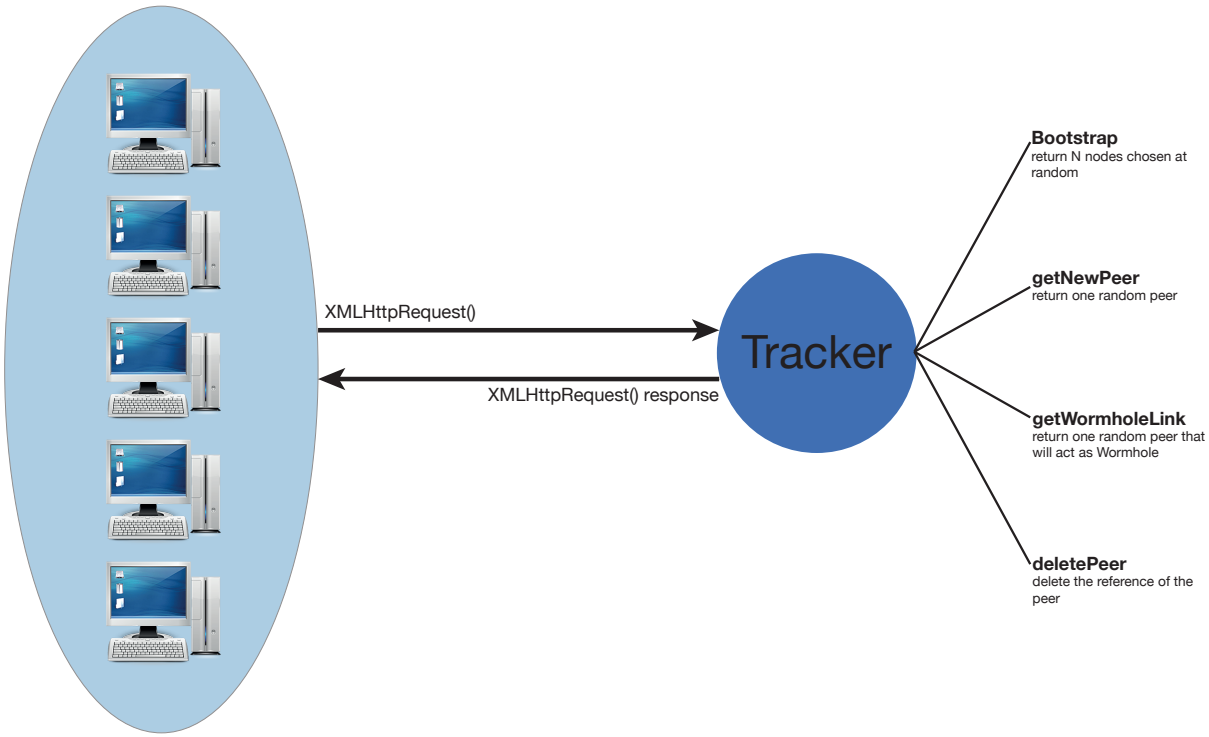

Figura 2.1: The main tasks of the tracker

3 Wormhole Peer Sampling Service

The Wormhole Peer Sampling Service (WPSS)[12] is an algorithm written by Roberto Roverso, Jim Dowling and Mark Jelasity published in 2013. It has been presented in the 13^{th} IEEE International Conference on Peer-to-Peer Computing. In the original paper it has been proved, compared to the state of the art at that time, that it has the same levels of samples' freshness of the other peer sampling services while the connection establishment rate is decreased by one order of magnitude. Another important feature is that it is a NAT-aware protocol, meaning that it handles situations in which some peers are behind NATs or firewall.

In the next sections we are going to see the reasons why these previous aspects might benefit our purpose, and how the algorithm really works.

3.1 Peer Sampling Service

A peer sampling service (PSS) is a service that runs on all the nodes in a distributed system and provides them with a uniform random sample of live nodes from all nodes in the system, where the sample size is typically much smaller than the system size.

The reason beyond this service is that, while in the past we had that the network was relatively small and all the nodes had the full view of it (it was almost a static network), now this is not the case anymore: in fact a general distributed system like a peer-to-peer system contains a lot of peers which can join, leave or crash at any time. In this system, having a complete view of the network could be useless or even impossible, so the peer sampling service provides a list of "fresh" nodes which reflects the peers known by the node inside the network.

A PSS can be implemented as a centralized service [10], using gossip protocols [11] or random walks [6]. Gossip-based PSSes have been the most widely adopted solution, as centralized PSSes are expensive to run reliably, and random walks are only suitable for stable networks [6]. However, in the Internet, where a high percentage of nodes are behind NATs, these traditional gossip-based PSSes become biased. Nodes cannot establish direct connections to nodes behind NATs (private nodes), and private nodes become under-represented in partial views, while nodes that do support direct connectivity, public nodes, become over-represented in partial views[4].

So to overcome the problem, a new class of NAT-aware gossip-based PSSes have appeared to be able to generate uniformly random node samples even for systems with a high percentage of private nodes, that is, nodes that reside behind a NAT and/or firewall.

State of the art NAT-aware gossip protocols, such as Gozar [4] and Croupier[7], require peers to frequently establish network connections and exchange messages with public nodes, nodes that support direct connectivity, in order to build a view of the overlay network. However, in commercial P2P applications such as Spotify [10], P2P-Skype [13] and as we saw in Sect. 1.1 Google's WebRTC, establishing a connection is a relatively complex and costly procedure. First of all because privacy is a concern, all new connections require peers to authenticate the other party with a secure server and to setup an encrypted channel. Another reason is that establishing a new connection may involve coordination by a helper service, for instance, to work around connectivity limitations that are not captured by NAT detection algorithms, e.g. using STUN Servers (Sect. 1.3).

In [12] they show that WPSS can provide the same level of freshness of samples as the state of the art in NAT-aware PSSes [7] but with a connection establishment rate that is one order of magnitude lower. This, without sacrificing the desirable properties of a PSS for the Internet, such as robustness to churn, NAT-friendliness and local randomness of samples.

3.2 WPSS: the algorithm

The main idea behind WPSS is that the service is separated into two layers. The bottom layer consists of a stable base overlay network that should be NAT-friendly, with private nodes connecting to public nodes, while public nodes connect to one another. The number of links in the base overlay is fixed at bootstrap-time and it is maintained during all the duration of the algorithm using the tracker (see Sect. 2) to replace the broken links with new ones.

On top of that there is the Wormhole overlay: every node periodically connects to a public node selected randomly from the base overlay (but not necessarily a neighbour). These links to random public nodes are called wormholes: each node systematically places samples of itself on nodes in the neighbourhood of their wormhole.

Fig. 3.1 shows this mechanism: in the bottom layer, the stable base overlay, there are private nodes connected to public nodes, and public nodes connected to one another; the upper layer, the wormhole overlay, shows a node with its wormhole placing samples of itself in the neighbourhood.

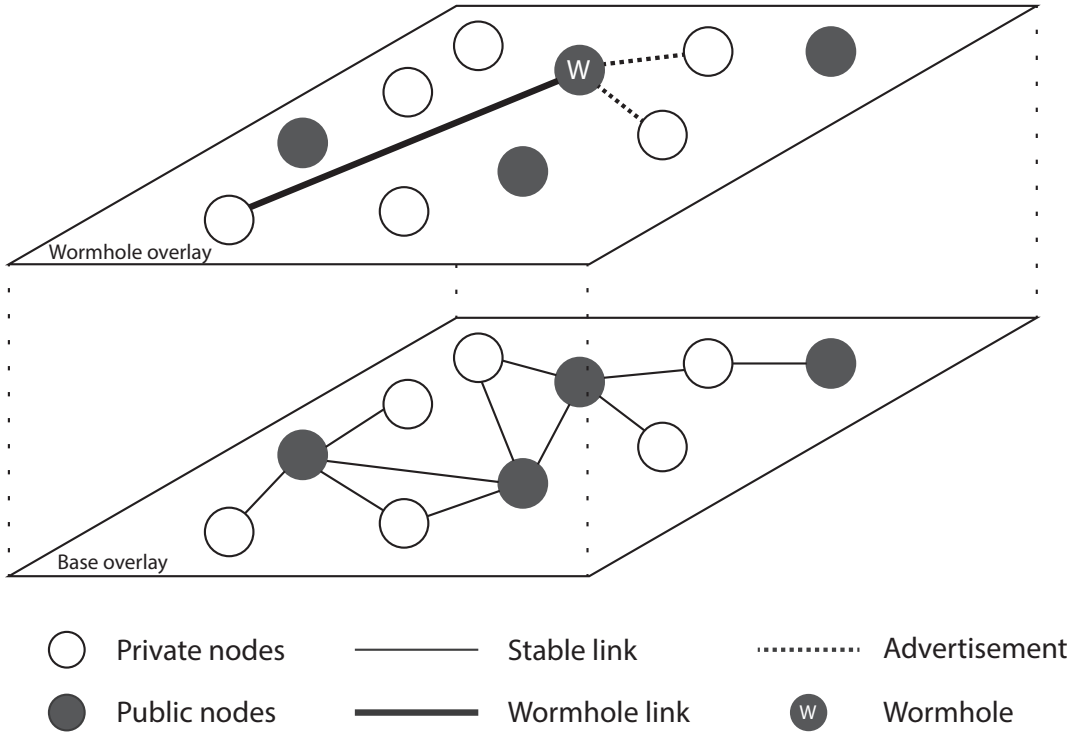

Figura 3.1: The difference between the two overlays

In other words, when the nodes start they ask the tracker the reference of N random public nodes: that will be their base overlay. Then, they periodically disseminate (push) advertisements of themselves over a small number of hops traversing the wormhole link and they place it at the node where this (typically short) walk terminates.

As explained before, a wormhole link points to a public node in the system that is selected independently at random. In WPSS every node discovers such a random public node to act as its wormhole, and every node only has one wormhole active at any given point in time. A new network connection will be created only when a wormhole is traversed for the first time by an advertisement. The wormhole is then reused for a few subsequent advertisements from the same initiator node, in which case no new network connection needs to be established. This makes it possible to decrease the number of new links established.

The very first time an advertisement reaches the wormhole, it can be placed at the public node as a new sample. However, if the public node already has a sample from the initiator node, then the advertisement will start a random walk over the base overlay until it either reaches a node that does not already have a sample from the initiator node or it reaches a given time-to-live (TTL)[12].

So, the first time a node will send the advertisement to its wormhole will create a link to it and it will place a sample first at that public node unless it already has an advertisement from the initiator. However, the reuse of the wormhole causes advertisements to continue, allowing them to also finish at private nodes, as private nodes are connected to public nodes over the base overlay.

When a wormhole is reused by an advertisement, the expected number of hops the advertisement will have to take increases, as it needs to reach a node that does not already have a sample from the initiator node. To counteract this, the WPSS defines a "wormhole renewal period" as a parameter for creating new wormholes, enabling users to control how frequently new network connections will be created. We will see in Sect. 5 that tuning this parameter is very important.

Algorithm 1: Wormhole peer sampling

```
upon wormholeFailure do
    wormhole ← tracker.getNewWormhole();
    connect(wormhole)
upon baseOverlayFailure do
    peer ← tracker.getNewPeer();
    connect(peer)
every Δ_wh = wormholeTimeout do
    disconnect(wormhole);
    wormhole ← tracker.getNewWormhole();
    connect(wormhole);
every Δ = adTimeout do
    ad ← createAd();
    ad.hops ← 1;
    sendAdToWormhole(wormhole, ad);
upon receivedAd(ad) do
    if ad.hop = getTTL() || acceptAd(ad) then
        view.addAd(ad)
    else
        neighbour ← getMetropolisHastingsNeighbour(baseOverlay);
        ad.hop ← ad.hop + 1;
        sendAd(j, ad);
```

In **Algorithm 1** there is the pseudo-code of the WPSS. The algorithm runs on all the peers (private and public) in the same way and contains three event-handlers and two timers.

The event-handlers are: *wormholeFailure*, *baseOverlayFailure* and *receivedAd*. The first two capture the failure of a node (respectively the wormhole and a node present in the base overlay). In these cases the node asks the tracker a reference of a new wormhole/node and it connects to it. The *receiveAd* will be explained later.

Then we have two timers: the wormhole renewal timer Δ_{wh} that triggers the generation of a new wormhole. In the function, the node disconnects from the old wormhole, it asks the tracker a new one and it connects to it. The other timer Δ is the rate at which advertisements are published. This timer triggers the sending of one advertisement over the local wormhole. In fact in the function the node creates a new advertisement setting the appropriate parameters (i.e. hop) and it sends it to its wormhole.

Finally the most important function: the *receiveAd*. This function is triggered when a node receives an advertisement: the task is to check if it has to consume it (hence adding it to its view) or it has to forward it.

The check is performed in two steps: first of all it looks if the TTL of the advertisement is reached, in such a case the node will necessary have to consume it; otherwise it checks if the sample is already contained in the view of the node, in this case it has to forward it. On top of that the *acceptAd* method makes sure that every node consumes advertisements at the same rate, namely one advertisement in each Δ time period. The reason is that the public nodes will receive a lot of advertisements in a

period (because the degree of these nodes are much higher than the private ones, and also because they probably are the wormholes of some private nodes), while the private nodes instead will receive few of them. This check is performed only on the public nodes, and it controls if the node has already consumed an advertisement in the period, in such a case it has to forward it, otherwise it consumes it. We will demonstrate in Sect. 5 that the *acceptAd* method successfully balances advertisements over public and private nodes.

If the node does not consume the advertisement, it sends it to another node using the Metropolis-Hastings transition probabilities over the (random, stable) base network[5]. Let d_i denote the degree of node i, that is, the number of neighbors of node i. The implementation of GetMetropolisHastingsNeighbor works as follows. First, we select a neighbor j with uniform probability, that is, with probability $1/d_i$. Then, we return j with probability $min(d_i/d_j, 1)$, otherwise we return node i itself, that is, the advertisement will visit i again[12].

4 Project Architecture

We implemented and tested the project in a real environment and also in simulation. WebRTC are web browser APIs, meaning that they can only work in a browser session. So we developed a first version of the WPSS that works in the browser, and tested it with few nodes (100) using Google Chrome[1] as browser. Then, in order to test it in a bigger environment we use a simulator. Both the versions are implemented using the framework **Hivejs-Framework** provided by the creator of the algorithm, the Hive Streaming[2] company. This framework is written in Typescript[3], and it enables us to choose if we want to run the project on browser or directly in *Node.js* simulating a lot of instances.

The Hivejs-Framework has a WebRTC module for handling peer-to-peer connections which, as explained in Sect.1.4, is EasyRTC.

[1] Google Chrome is a free-ware web browser developed by Google

[2] Hive Streaming provides network solutions for media distribution and performance analysis. Started as a spin-off in 2007 from the Swedish Institute for Computer Science and the Royal Institute of Technology in Stockholm, the company maintains a strong focus on research and development. `https://www.hivestreaming.com`

[3] TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. `http://www.typescriptlang.org`

5 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/infrastructure/.

[4] J. Dowling A. Payberah and S. Haridi. "gozar: Nat-friendly peer sampling with one-hop distributed nat traversal". In *Distributed Applications and Interoperable Systems*, 2011.

[5] S. Chib and E. Greenberg. "understanding the metropolis-hastings algorithm". *The American Statistician*, 49(4):327–335.

[6] N. Duffield S. Sen D. Stutzbach, R. Rejaie and W. Willinger. "on unbiased sampling for unstructured peer-to-peer networks". In *IEEE/ACM Transactions on Networking*, August 2009.

[7] J. Dowling and A. H. Payberah. "shuffling with a croupier: Nat-aware peer-sampling". In *Proceedings of The 32nd Intl. Conf. on Distributed Computing Systems (ICDCS 2012)*, pages 102–111, Los Alamitos, CA, USA: IEEE Comp. Soc., 2012.

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

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

[10] G. Kreitz and F. Niemela. "spotify – large scale, low latency, p2p music-on-demand streaming". In *Tenth IEEE Intl. Conf. on Peer-to-Peer Computing (P2P'10)*, August 2010.

[11] R. Guerraoui A.-M. Kermarrec M. Jelasity, S. Voulgaris and M. van Steen. "gossip-based peer sampling". In *ACM Transactions on Computer Systems*, August 2007.

[12] Jim Dowling Roberto Roberso and Mark Jelasity. *Through the Wormhole: Low Cost, Fresh Peer Sampling for the Internet.* 13-th IEEE International Conference on Peer-to-Peer Computing, 2013.

[13] N.Daswani S.Guha and R.Jain. "an experimental study of the skype peer-to-peer voip system". In *Proceedings of IPTPS*, page pp. 1–6, Santa Barbara, CA, February 2006.

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