Introducing ufo.js: A browser-oriented p2p network

Conference Paper · February 2014

DOI: 10.1109/ICCNC.2014.6785359

CITATIONS
7

READS
416

3 authors:

Antonio Bevilacqua
University College Dublin
15 PUBLICATIONS **169** CITATIONS
SEE PROFILE

Pasquale Boemio
4 PUBLICATIONS **36** CITATIONS
SEE PROFILE

Simon Pietro Romano
University of Naples Federico II
156 PUBLICATIONS **953** CITATIONS
SEE PROFILE

Introducing *ufo.js*: a browser-oriented p2p network

***Abstract*—In this paper we present ufo.js, a novel network architecture enabling the development of browser-based peer-to-peer web applications. ufo.js leverages state-of-the-art technologies in the field of real time communications in the web and provides programmers with the functionality needed in order to embed novel peer-to-peer applications directly into web browsers.**

The idea of ufo.js starts when the W3C WebRTC working group begins to develop the so-called *datachannel* API. This new interface allows two web browsers to establish a communication channel on top of which it is possible to send either raw data or strings. Ufo.js uses the datachannel as the default communication means between any pair of peers and hence represents a valid alternative to classic client-server desktop solutions.

The article describes the design, implementation and deployment of an ufo.js network, while also presenting the results of a test campaign aimed at assessing both its performance and potential overhead.

***Index Terms*—peer-to-peer architectures, datachannel API, real-time communication in the web, webrtc, javascript, node.js, websocket**

I. Context and Motivations

Web applications have been laid on the classical client-server paradigm since they saw light. This is due to the absence of appropriate technologies supporting direct communication between web browsers and clearly entails the inability to develop web applications based on peer-to-peer protocols. Among the applications which are most impacted by such an inability, real-time multimedia systems and file sharing architectures definitely play a major role. In the depicted scenario, developers are forced to implement dedicated desktop or mobile applications, without relying on any form of support from web browsers. Though, support for enabling real-time communication in the Web is currently gaining momentum with the two main Internet standardization bodies, the IETF and W3C [1]. Web Real-Time Communication (WebRTC) is an upcoming standard that aims to enable real-time communication among Web browsers in a peer-to-peer fashion. The IETF RTCWeb and W3C WebRTC working groups are jointly defining both the APIs and the underlying communication protocols for setting up and managing a reliable communication channel between any pair of next-generation Web browsers.

WebRTC core development is currently focused on multimedia communications and has motivated big players (like, e.g., Google, Mozilla and Ericsson) in investing significant resources in this research and engineering effort. Besides multimedia, the youngest segment of WebRTC technologies deals with end-to-end data transmission. Specifically, WebRTC provides an abstraction for a communication channel, named *datachannel*, on top of which it is possible to send and receive both binary (also known as *blob*) and string data.

We made use of the above mentioned WebRTC data tools, in conjunction with some other standardized technologies such as websockets and the NPAPI (Netscape Plugin API) libraries, in order to develop a novel network architecture called *ufo.js*, whose purpose is to provide a complete platform on top of which it is possible to implement any kind of data-centered peer-to-peer web applications.

The paper is composed of six sections. In section II we describe our architecture for peer-to-peer data communications in the web, by also discussing issues associated with topology management and components deployment and configuration. Section III presents some practical aspects related to network implementation and deployment. Performance figures are illustrated in section IV, whereas section V provides a few pointers to related works in the field of peer-to-peer, web-based and data-oriented communication. Finally, section VI illustrates concluding remarks and identifies the main directions of of our future work.

II. Design

In this section, we describe all the architectural details underlying the ufo.js network. These involve structural issues (network topology and organization), as well as behavioral issues (bootstrap procedure and network conduct).

A. Overview

The ufo.js project has been conceived at the outset by carving into the stone the following fundamental architectural constraints:

100% **browser-oriented** : ufo.js is a fully-fledged platform on top of which it is possible to implement peer-to-peer web applications;

ready to use : ufo.js makes available to the programmer a comprehensive development stack, hence not requiring any additional components during the implementation phase;

lightweight : ufo.js minimizes both the number of messages flowing across the network and the amount of network information stored at each peer.

The mentioned characteristics make ufo.js a valid choice over classic desktop solutions for peer-to-peer based applications development.

B. Components

Ufo.js is a hybrid peer-to-peer network in which we can identify three different entities:

node : a node is a web page opened inside a browser. Every node keeps `CP_SIZE` connections with other nodes, up to a threshold value of `MAX_CP_SIZE`. A node is said to be *internal* if

$$\texttt{CP\_SIZE} = \texttt{MAX\_CP\_SIZE} \tag{1}$$

If instead

$$\texttt{CP\_SIZE} < \texttt{MAX\_CP\_SIZE} \tag{2}$$

that node is said to be *external*. The set of active connections is named *connection pool*.

supernode : a supernode is a node providing an entry point for all the nodes that will bootstrap on the network. When a node becomes a supernode, it has to publish its address in order to make it available to all potential peers.

optional server : in this scenario it is possible to add a central server that simplifies the publishing procedure for supernodes and provides all other nodes with a list of public supernodes addresses. Such a server can also make available application related web pages.

Hereinafter we will consider a full ufo.js implementation including all of the above mentioned components.

C. Network construction

Nodes and supernodes within the network are arranged as a connected graph with no orientation. This is achieved through three different operations: *rise*, *bootstrap* and *densify*.

During the rising phase, a node publishes an address where it can be contacted by anyone willing to join the peer-to-peer overlay. If the node in question is already connected to a ufo.js network, it will make available a new entry point; otherwise, it will spawn a brand new network. A node accomplishes the rising phase by registering its address at an auxiliary server through an HTTP POST request.

As shown in figure 1, the bootstrap operation can be split into several steps. Let P_1 be a web browser outside the network, SN a supernode already inside the network and S a subsidiary server.

Phase 1 : P_1 sends an HTTP GET request to S, S replies with the main application page, as well as a cookie containing an `id`.

Phase 2 : P_1 selects the address of SN in the list contained inside the received application page, and sends to SN a peering request.

Phase 2.1 : SN receives the peering request and checks the size of its connection pool. If `CP_SIZE` is less than `MAX_CP_SIZE`, SN adds P_1 to its connection pool and sends back to it a peering reply. Otherwise, SN forwards the peering request to a different supernode inside the network.

If P_1 receives a peering reply before a predefined timeout occurs, it opens a datachannel towards the originator of the reply. This ends the bootstrap procedure.

In order to improve the robustness of the network, each node may send new peering requests if its connection pool respects the following condition:

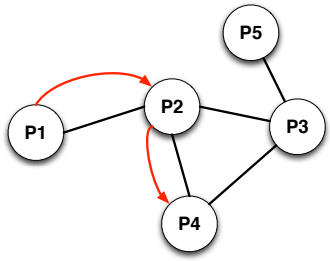
$$\texttt{CP\_SIZE} + 1 \leq N < \texttt{MAX\_CP\_SIZE} \tag{3}$$

Each external node has to accept a peering request if the originator of the request itself is not already present inside the connection pool. Otherwise it has to forward that request. This operation is called densification.

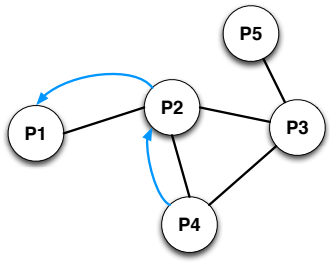
Let us consider the scenario in figure 2, with

$$\texttt{MAX\_CP\_SIZE} = 3$$
$$N = 2$$

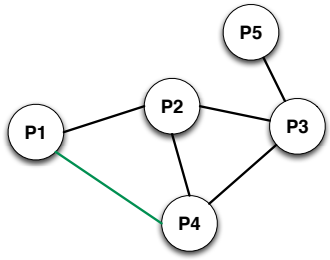
In figure 2(a) P_1 holds just one active connection. It then sends a new peering request to P_2. P_2 is an internal node and, according to the densification rules, forwards the request to P_4. P_4 is an external node not directly connected to P_1. It



(a) Densify request



(b) Densify reply



(c) Densify completed

Fig. 2. Densification phase

then has to accept the request and send its reply backwards to P_1, as shown in figure 2(b). Eventually, P_1 and P_4 establish a new connection, as shown in figure 2(c).

D. Routing

Routing over an ufo.js network requires that each node is provided with a unique `ID`. In the example shown in paragraph II-C, such an `ID` is created by the subsidiary server.

Every packet flowing through the network has the format shown in figure 3, where:

body is the actual payload of the packet,
type is the payload type,
path is the set of `ID`s of the nodes crossed by the packet,

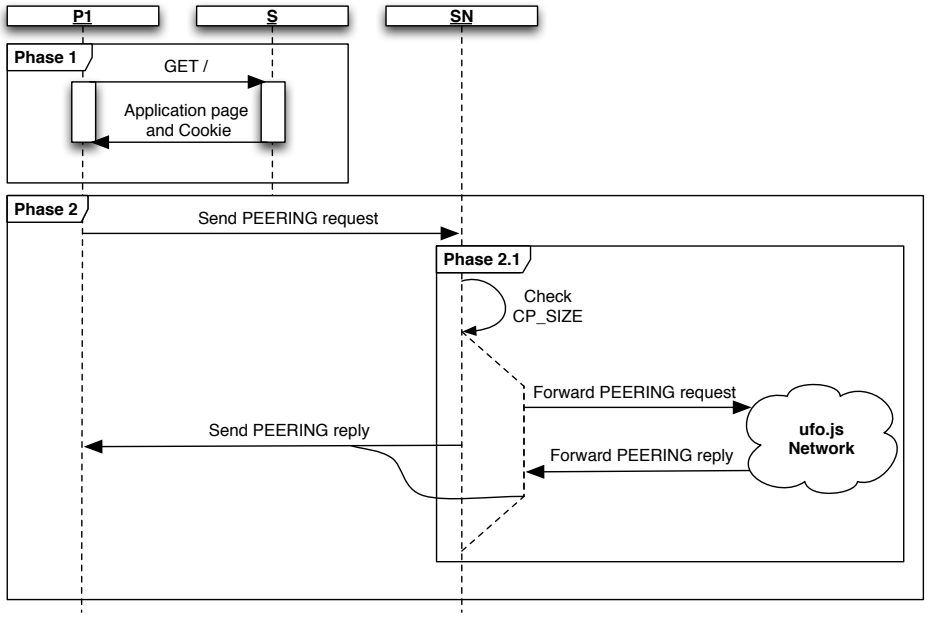


Fig. 1. Bootstrap phase

```
{
 type : 'type',
 body : { obj },
 path : [ ... ],
 isBoozer : true/false
}
```

Fig. 3. Packet structure

isBoozer is a boolean flag representing packet transmission mode.

Packets can be sent either in *discovery* or in *direct* mode.

A packet sent in discovery mode neither knows its destination `ID` nor needs to care about it. Discovery packets proceed through the network along a random path. Each node selects the next hop of a discovery packet by choosing randomly within its own connection pool, excluding all the nodes already contained into the path field of the packet. If all the nodes in the pool are already present inside the path, the packet must be sent back to the previous hop. Just before sending the packet, the current node adds its `ID` to the path field. In discovery mode, the above mentioned *isBoozer* flag has to be set to true.

A packet sent in direct mode is destined to a specific `ID`. A direct packet contains the path to be followed inside its own path field. Every node learns the next hop by reading and removing its `ID` directly from within the packet. A packet arrives at destination when its path field is empty. In direct mode, the *isBoozer* flag must be set to **false**.

In figures 4 and 5 we report the flow diagrams associated, respectively, with events transmission and reception inside a node.

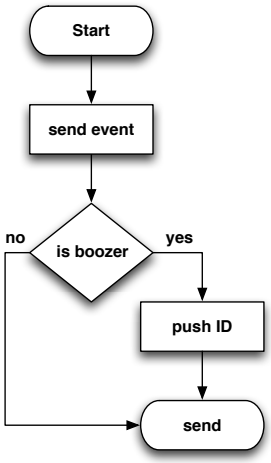


Fig. 4. Packet transmission flow

III. Implementation

We have developed a fully-fledged javascript implementation of ufo.js, composed of:

- ✓ an auxiliary server providing the HTML application pages and managing the list of supernodes;
- ✓ an NPAPI (Netscape API) plugin enabling web browsers to listen to incoming websocket connections;
- ✓ a set of javascript classes granting the needed peering functionality to web browsers.

In the next sections we analyze in some detail all of the above listed components, by focusing on the adopted technologies, as well as on how they were employed in ufo.js.

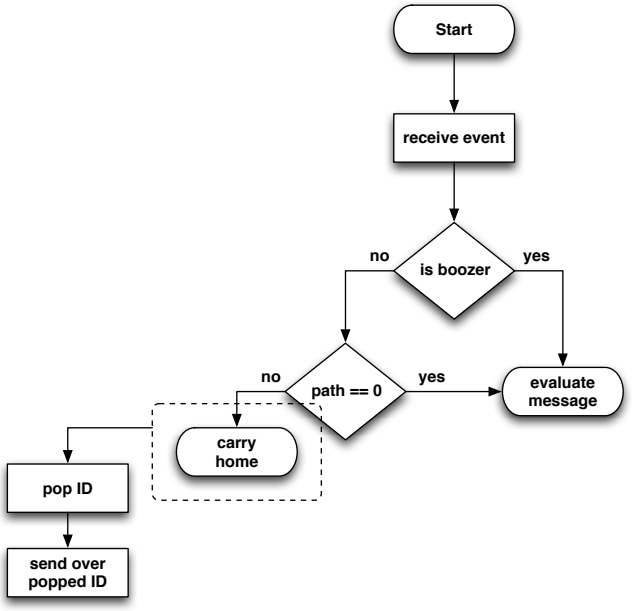


Fig. 5. Packet reception flow

A. Overview

The idea of ufo.js starts when the WebRTC working group begins to develop the *datachannel* API. This new interface allows two web browsers to establish a communication channel on top of which it is possible to send either raw data or strings. The actual channel allocation is completed as soon as the *signaling procedure* is over. During the signaling procedure, web browsers exchange two SDP (Session Description Protocol) messages, as mandated by the so-called *offer/answer* model. Both messages provide a description of the session being opened and carry fundamental information, including, among other things, *ICE candidates* and cryptographic algorithm in use.

Ufo.js uses the datachannel as the default communication channel between any two peers. Offer and answer messages are properly minimized and then exchanged through peering request and related peering reply packets. Both the peering packet and the peering reply packet contain all the essential fields required to properly construct the original SDP string, as well as an originator field carrying the `ID` of the node that generated it.

Figure 6 shows the structure of an ufo.js packet whose body field contains a peering packet. An ufo.js packet carrying a peering packet must have its type field set to **peering** and its isBoozer flag set to **true**. A node sending a peering request doesn't know a priori which peer node will accept it. Though, such a node exists for sure in any case, due the previously described mechanism based on bootstrap and densify procedures. For this reason, peering packets are sent in discovery mode.

Figure 7 sketches the structure of an ufo.js packet whose body field contains a peering reply packet. An ufo.js packet carrying a peering reply packet must have its type field set to **peeringReply** and its isBoozer flag set to **false**. A node

```
{
 type : 'peering',
 body : {
          originator : 'originatorID',
          PORT_NUMBER : 'port_number',
          EXTERNAL_ADDR : 'external_address',
          CANDIDATE_11 : 'candidate_11',
          ...
        },
 path : [ ... ],
 isBoozer : true
}
```

Fig. 6. Peering packet

sending a peering reply packet is made aware of the destination node by reading its `ID` in the originator field inside the previously received peering packet. The path to be followed has certainly been carved in the path field of such a packet.

```
{
 type : 'peeringReply',
 body : {
          originator : 'originatorID',
          PORT_NUMBER : 'port_number',
          EXTERNAL_ADDR : 'external_address',
          CANDIDATE_11 : 'candidate_11',
          ...
        },
 path : [ ... ],
 isBoozer : false
}
```

Fig. 7. Peering reply packet

B. Auxiliary server

We developed an auxiliary server by leveraging the Node.js framework. Such a server is therefore entirely written in javascript and uses Redis to manage supernodes information. The current implementation of the auxiliary server supports two operations:

GET on *`/nodepage.html`* : when the server receives a GET request on */nodepage.html*, it generates a random string and retrieves from its cache the list of active supernodes. The generated string is added to the `Set-Cookie` header field of the HTTP response, while the supernodes list is included inside the HTML application page.

POST on */serverize* : when the server receives a POST request on */serverize*, it takes the requestor `ID` from the cookie and stores its location information inside the cache. If the requesting `ID` is already present inside the cache, the server simply updates such information.

C. Supernode plugin

We used the C++ NPAPI/ActiveX libraries in order to create a plugin suitable for all browsers. This plugin allows a node to become a supernode by enabling it to open a web socket in passive mode.

D. Browser side classes

We developed a collection of javascript classes that implement all the routing and network management algorithms. These classes are compressed in a bundle and included inside the application page.

IV. Tests

For our tests, nodes and supernodes run on Macbooks Pro 2.4 GHz Intel Core 2 Duo with OS X version 10.8.2. We also use a Raspberry PI ArmV6 with Arch Linux version 3.6.11-4-ARCH+ (shown in figure 8) as publishing server, running Node.js version 0.8.18 and Redis version 2.6.8. Ufo.js has been deployed on such a server.

The entire work we have presented, as well as the experimental campaign herein described, are targeted for Google Chrome Canary and Mozilla Firefox Nightly. In particular, as of 5th April 2013:

- ✓ Google Chrome, version 27.0.1416.0 canary, brings a quite limited implementation of datachannels. As a matter of fact, it is not yet possible neither to create reliable datachannels nor send binary data. Therefore, we are not able to send messages whose payload is bigger than an MTU (about 1200 bytes); all out-of-bound messages are dropped without errors, exceptions or warnings. However, Google Chrome Canary shows a very stable and reliable behavior during both its ordinary usage and our stress test campaign. We did not experience any crashes or similar types of failures.
- ✓ Mozilla Firefox, version 21.0a1 nightly, contains an advanced implementation of datachannels. Within Firefox, we can create reliable datachannels and use them to transfer any kind of information (including binary data) with varying message sizes. Though, Firefox datachannel implementation is not the final and stable one: some methods are still in their draft version, while others are just temporary (e.g., the `connectDataConnection` function, that will soon disappear). Moreover, Firefox exhibited a temperamental behavior with occasional crashes during our tests with datachannels.

The ufo.js implementation here under test uses the above mentioned datachannel version, currently in hard beta. Therefore, we decided to conduct a preliminary test campaign aimed to evaluate datachannel performance in the presence of the two mentioned browser implementations.

A. Datachannel

As a baseline, to assess pure datachannel time performance we make use of a demo page whose only purpose is to instantiate and connect a datachannel between two peers. During this test, as shown in figure 9, we evaluate potential differences between Chrome (blue line) and Firefox (red line) regarding the implementation of the standard datachannel API. It clearly comes out that, although Firefox shows a slightly slower trend, both browsers ensure a quite constant datachannel creation/connection time.

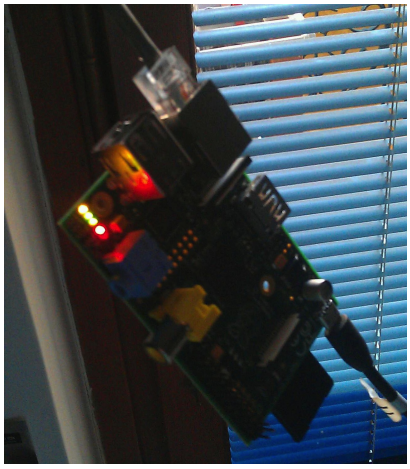


Fig. 8. Ufo.js test server

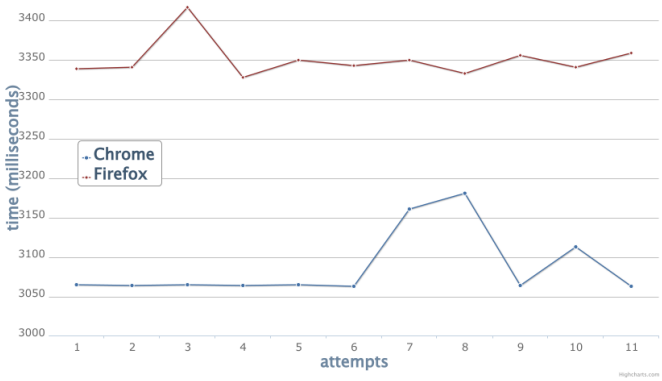


Fig. 9. Datachannel allocation time

B. Memory occupation

We further investigate memory consumption at a supernode by exploiting the profiling features made available by Chrome Developer Tools. Let N_1 be the node under test. The testing procedure runs as follows:

1) N_1 contacts the auxiliary server and gets the page shown in figure 10;
2) N_1 clicks on the *Increase cPool* button, setting its MAX_CP_SIZE to 9999;
3) N_1 clicks on the *Serverize* button, activating the supernode plugin and hence becoming a supernode;
4) a number of simple nodes contact N_1 and open a datachannel towards it. This is done by pressing on the page of each node the *Test* button, that creates 10 more nodes and connects them to N_1.

We take a heap snapshot every time CP_SIZE of N_1 increases by 10, starting from 0 and up to a threshold of 110. The total amount of memory required by the application starts from 1.9 Mb for 10 nodes and increases roughly by 30 Kb every 10 nodes, up to 2.7 Mb in the presence of 110 nodes, as shown in figure 11.

Figure 12 reports the connection pool memory occupation with blue bars, while the red line indicates its occupation as a percentage of the total busy memory.

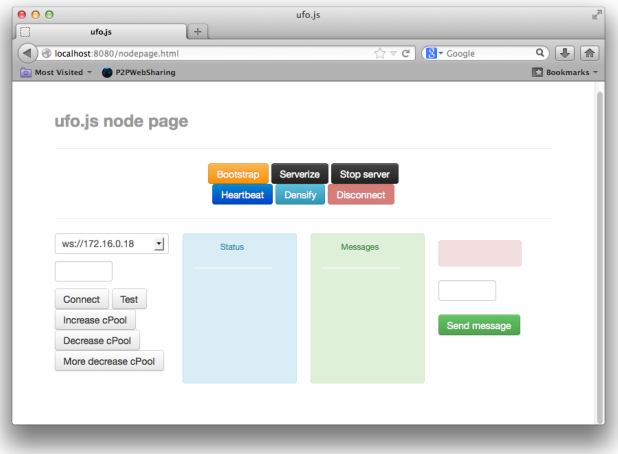


Fig. 10. Ufo.js application page

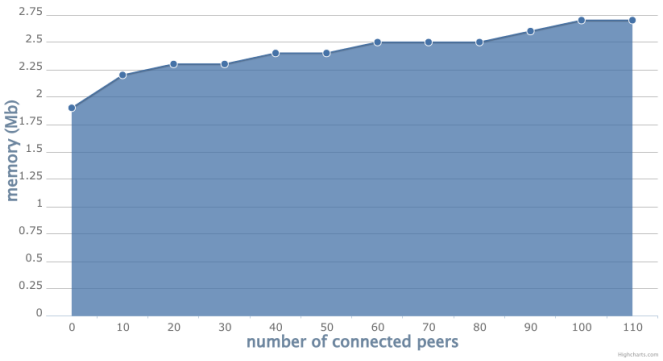


Fig. 11. Ufo.js application memory consumption

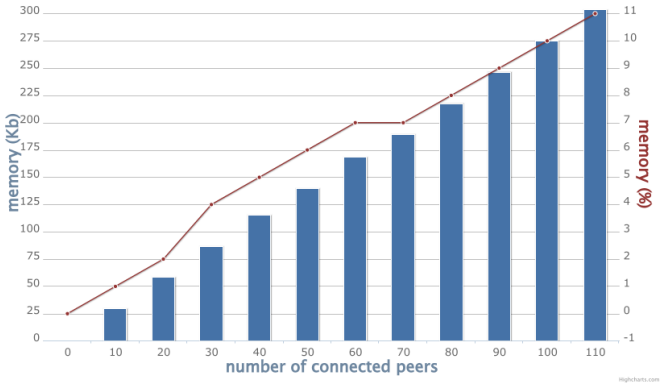


Fig. 12. Supernode connection pool memory occupation

TABLE I
NETWORK BOOTSTRAP TIMES

NAME	LEVEL	TIME (ms)
DzY	1	673
x4z	1	648
jMK	1	369
oXI	1	618
vuA	2	734
W2j	2	366
r4Z	2	498
aH1	2	566
cuf	2	447
cNj	2	697
tvP	2	562
vjN	2	840
Ynn	2	847

C. Normal network usage

In this last section we evaluate the average behavior of the overall system by looking at how nodes join and alter the network. This test runs as follows:

1) We open the application page shown in figure 10 and click on the *Serverize* button, hence creating a new supernode;
2) for each new node to be inserted in the network, we open the application page in a new window, select the address of the supernode within the combo box on the left and click on the *Bootstrap* button.

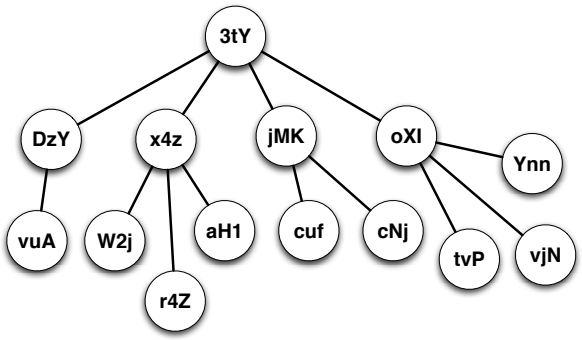
In this scenario no changes are committed to the nodes and supernodes creation procedure. Therefore, MAX_CP_SIZE is always equal to 4. After the bootstrap phase, the network looks like the one in figure 13(a) (node IDs are properly contracted from 16 to 3 characters in order to make the figure more readable). Nodes are arranged as a tree with a maximum depth of 2 levels. In particular, 4 level-1 nodes are connected to the supernode (3TY) and 9 level-2 nodes follow. In table I we report the bootstrap times for every node.

After the bootstrap phase we launch the densification procedure for every node that allows it. In particular, all level-2 nodes, as well as DzY, can do it.

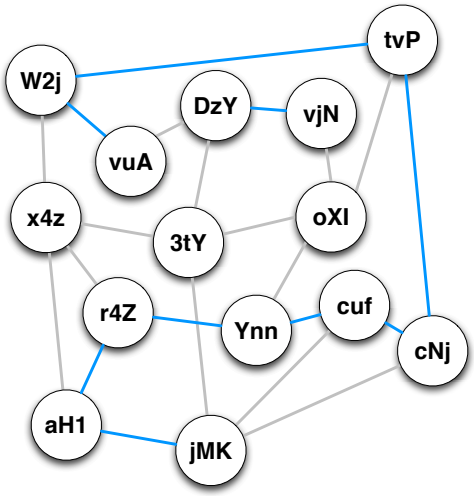
As an example, let us consider node vuA, whose densification procedure progresses throughout the following steps:

- ✓ vuA sends a densification request to DzY;
- ✓ DzY checks its connection pool and finds vuA is already present in it;
- ✓ DzY forwards the request to 3TY;
- ✓ 3TY checks its connection pool size and observes it is already full;
- ✓ 3TY randomly picks a node from its pool. It chooses x4z and forwards the request to it;
- ✓ x4z, for the same reason as 3TY, forwards the request to the randomly chosen node W2J;
- ✓ W2J can accept the request and replies to vuA allowing the creation of a new connection.

In figure 13(b) we show a possible structure of the network after all the nodes who were eligible for the densification procedure have completed it. Light blue links are those created with densification. As we can notice by comparing figures 13(a) and 13(b), the network structure moves from a tree-shaped graph to a connected graph with no particular shape. This behavior provides the network with a high level of reliability by creating new paths connecting nodes and hence becoming more robust to node failures.



(a) Network after bootstrap



(b) Network after densification

Fig. 13. Network creation use case

V. Related work

The datachannel technology is a very powerful one and can potentially impact the way we think of the entire web application paradigm. Though, the datachannel APIs currently provide just a partial (and in any case unstable) implementation. This entails that applications based on these emerging APIs cannot be considered other than beta releases, if not just proof-of-concept demonstrations.

Companies which are currently leading the development and browser integration efforts associated with the datachannel, such as Google and Mozilla, do provide some demo pages allowing users to test the state of the art of the above mentioned APIs. Google Chrome's test page can be found at [2], while Mozilla Firefox testing site is reachable at [3].

Among real life applications using the datachannel as best as they can, we should definitely mention *shareit* [4] and *sharefest* [5]. Such applications allow browsers to share files in the absence of any form of upload to external servers, as opposed to well-known services such as *Dropbox* or *Google Drive*. Both *shareit* and *sharefest* use an external server holding and managing a connection to each peer; these connections are used to accomplish all the signaling procedures between peers. Thus the actual p2p communication happens during file transfers.

As we can infer, all of the above mentioned applications introduce a user-friendly front-end towards the datachannel APIs. Their main focus is therefore to provide browsers with fairly scalable file sharing capabilities.

On the other hand, ufo.js aims to create a *pure* p2p overlay network exploiting all the potential of datachannel-aware browsers. The challenge is to let this approach become widely available and hence represent a real opportunity for the development of advanced p2p applications that are unaware of the inner p2p mechanisms adopted, but just focus on the specific business logic.

VI. Conclusions and future works

In this paper we presented ufo.js, a web stack that makes it possible to develop advanced web applications based on a peer-to-peer communication paradigm. Ufo.js takes care of both routing and signaling, hence providing self-organizing networking mechanisms based on a pure peer-to-peer approach. We discussed the main issues we had to face while designing and implementing our framework, which leverages state-of-the-art achievements in the field of real-time communication in the web. A preliminary set of tests has been conducted in order to assess the performance achieved by the currently available browser implementations of the so-called datachannel API. Then, a thorough experimental campaign has been carried out with the aim of evaluating the overhead introduced by our p2p javascript library, in terms of network topology setup time (involving a bootstrap phase followed by a densification phase), as well as memory consumption. We do believe that the architecture we propose unveils an unprecedented potential to all web developers who have a stake in the design of innovative peer-to-peer applications and who are looking at the recent WebRTC/RtcWeb standardization efforts as the necessary missing brick. As to our future work, we plan to stabilize the current implementation of ufo.js by making it more reliable and even lighter, with the final objective of using it as a structural layer for file sharing and multimedia serverless web applications.

References

[1] S. Loreto, and S. P. Romano. Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts. Internet Computing, IEEE , vol.16, no.5, pp.68-73, Sept.-Oct. 2012. doi: 10.1109/MIC.2012.115.

[2] Google Chrome demo page. [Online]. Available: http://webrtc.googlecode.com/svn/trunk/samples/js/demos/html/dc1.html

[3] Mozilla Firefox demo page. [Online]. Available: http://mozilla.github.com/webrtc-landing/data_test.html

[4] Jesús Leganés Combarro, "ShareIt!". [Online]. Available: http://shareit.piranna.5apps.com/

[5] Peer5, "Sharefest". [Online]. Available: http://sharefest.peer5.com/