Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Implementation of a Browser-based P2P Network using WebRTC

**Max Jonas Werner, Christian Vogt**

**Research Report**

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Max Jonas Werner, Christian Vogt

# Research Report

Implementation of a Browser-based P2P Network using WebRTC submitted in the context of Projekt 1

in the course Master of Science
at the Department of Computer Science
at the Faculty of Engineering and Computer Science
of Hamburg University of Applied Sciences

Submitted on: 28 Jan 2014

# Contents

# 1 Introduction

The Web is since its incarnation formed by classic client/server architectures using the HTTP(S) protocol. There are several use cases, though, where a peer-to-peer (P2P) approach is preferable. Additionally, the reliance on servers that users have no control over can pose a security and privacy risk for sensitive data.

A set of new web technologies called WebRTC is currently under development to enable a real browser-to-browser communication channel (see figure 1.1). WebRTC enables web applications to establish a direct communication channel between two browsers without relaying the data through a web server. It consists of an API [1] defined by the W3C and a set of underlying protocols defined by the IETF Rtcweb Working Group [2]. The possibility of establishing peer-to-peer channels between two browsers and the expected broad deployment (a browser is installed on most current consumer devices from PCs to phones to TV devices) opens the opportunity for new use cases that were only possible until now by directing all traffic through a central server or by using proprietary technology plugins like Adobe Flash.

Such use cases range from simple real-time P2P audio/video chats to browser-based content delivery networks, streaming audio/video or file sharing to virtual whiteboards or collaborative editing of documents. More complex use cases involve asynchronous server-less content publishing and consumption that we will explore in detail below.

Currently, Google and Mozilla are working intensely on implementing WebRTC in their browsers Chrome and Firefox, respectively. Many of the specified features already work in stable or beta versions of either browser. Still, though, the specification process still continues. As such, the specification details are likely to change in the next month. These details include but are not limited to supporting audio/video codecs (mostly a matter of non-technical arguments such as potential patent claims) or feature groups such as identity verification of peers and bundling media streams.

At the heart of WebRTC lies the possibility of establishing audio/video channels using the Secure Real-time Transport Protocol (SRTP) and data channels using the Stream Control Transmission Protocol (SCTP) over Datagram Transport Layer Security (DTLS) over UDP. Such data

channels can exchange arbitrary binary or string-type data (see figure 1.2) and serve as the foundation of our work.

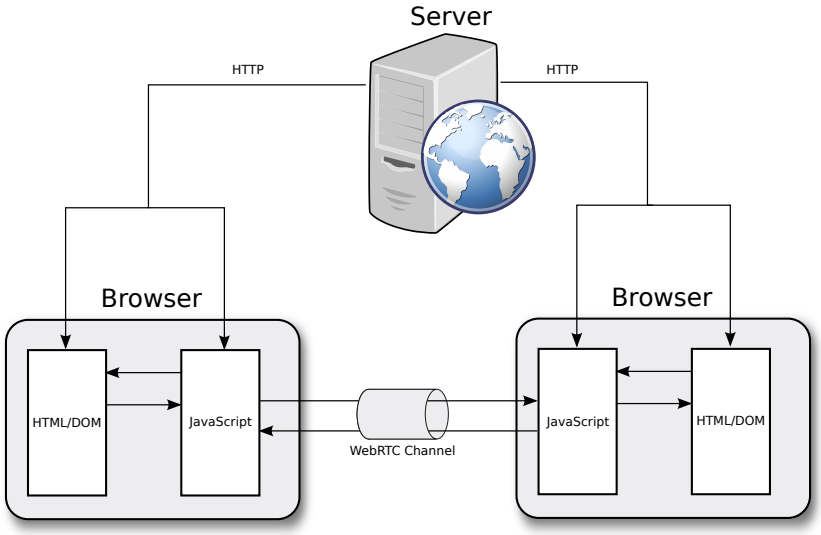


Figure 1.1: Schematic view of a WebRTC connection. The server delivers applications (consisting of HTML, CSS and JavaScript code as well as resources such as images) to all the client browsers. Additionally, the server handles the signaling of a WebRTC connection, therefore serving as a central connection establishment entity. After the inital connection establishment, the server may be shut down.

In this project report, we present the results of our effort to implement a generic user-centric content-sharing facility using WebRTC. Our first step to reach this goal was to implement a Peer-to-Peer network using WebRTC as underlying transport technology as outlined in [3]. WebRTC Data Channels [4] allow for the secure transfer of generic data (text, binary data) from one browser to another. They comprise the main transfer mechanism used to build the P2P system described in this report. On top of these Data Channels, we implemented a protocol suitable for joining the WebRTC network and for maintaining connections between clients.

The resulting JavaScript library provides an API for applications to store content in and retrieve content from the underlying P2P network; it can be used as a drop-in for existing web applications. On top of this library, we implemented demo applications that serve as simple use cases and as a measurement for the ease of the API of the core library.

The remainder of this report is organized as follows. We define the problem scope in detail and outline related work in chapter 2. The architecture of our proposed solution is explained in chapter 3 and the intermediate steps and results of our implementation effort is discussed in chapter 4. There we describe the API of our publishing library, while chapter 5 depicts a number

of demo applications built on top of the resulting core library. In chapter 6 we draw a conclusion and give an outlook on future work.

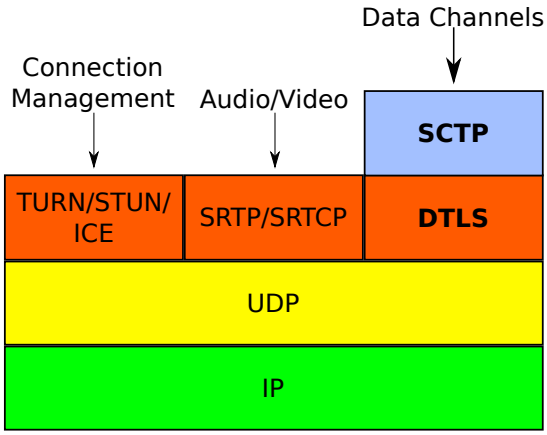


Figure 1.2: The protocol stack of WebRTC is divided into a connection management component for establishing and maintaining connections even across middle boxes, an A/V component and a Data Channel component. Both audio/video data as well as Data Channel streams are encrypted end-to-end using SRTP/SRTCP and DTLS, respectively.

# 2 Browser-based Publishing and Related Work

## 2.1 Problem Statement

With the uprise of Web 2.0 technologies over the past ten years, Web platforms have shifted from pure content silos to services for publishing user-generated content. Today, users also see the Web as a platform to share media, documents and exchange individual information such as instant messages among each other. Currently, perceiving user-generated content on the Web follows a centralized, host-based approach. Examples for such central content sharing community platforms are Facebook, Flickr and Youtube. Publishing content on the Web thus requires access to infrastructure such as Web servers and name resolution services like DNS. Content addresses (HTTP URLs) are tightly bound to specific hosts via their DNS name.

The concept of Information-centric Networking (ICN) [5] approaches the problem of content dissemination on the network layer and elevates the meaning of content by assigning explicit identifiers to data rather than referring to the location of content, e.g. by using an HTTP URL. Content can be stored directly in the network which inherently provides functionality to store and cache it. The ICN approach, however, suffers from conceptual vulnerabilities as Wählisch et al. [6] have pointed out. Besides unresolved security issues, every content publication act requires modifications of the control plane. As such, the control plane is inadvertently opened up towards end users.

In contrast to ICN, our approach is not supposed to operate on Internet-scale and runs on the application rather than the network layer. It provides an infrastructure-independent name and content access architecture. Web application providers could potentially benefit from such a system by reducing both the bandwidth consumed on the server side as well as transfer delays. The clients, on the other hand, benefit by not being required to rely on a server for sharing content with other users. As the browser is the natural application platform for the Web, we want to leverage its broad deployment and operating system independence. WebRTC provides the required transport mechanism by offering the Data Channel protocol that is used to transfer generic data directly between two browsers.

## 2.2 Related Work

Various approaches to ICN have been proposed that provide efficient user-centric publishing mechanisms [7]. For addressing content, there exist two major techniques: Using either a flat identifier or a naming hierarchy. The Data-oriented Network Architecture (DONA) [7] is an example of an architecture that uses flat identifiers. The DONA approach assures self-certifying names. Hierarchical identifiers on the other hand, that are used for example in Content-Centric Networking (CCN) [7] allow for content aggregation on intermediate routers. Additionally, such namespaces allow for wildcard searches.

Research on leveraging native browser technologies for content distribution has already been addressed by Zhang et al. [8] introducing an implementation of a browser-based content delivery network (CDN). The authors have investigated the possibilities of building a CDN service that is based on a centralized P2P network using the Flash plugin provided by Adobe. Their implementation is centered around a coordinator node that holds mappings between peers and the data stored on these peers. Every distributed asset in the P2P network is fetched using JavaScript from one of the participating peers. Taking advantage of this solution requires the modification of the HTML code and the setup of the controller.

Meyn [9] examines a way to distribute the load and stream video content between browsers using WebRTC, thus reducing the bandwidth cost of content providers. The author uses a BitTorrent-like architecture involving a tracking server for discovering content. The integration of WebRTC into the current SIP ecosystem for real-time conferencing is evaluated in [10].

Despite ongoing research efforts with regards to WebRTC and Browser-based content dissemination, most research focuses on the audio/video capabilities of WebRTC, using SRTP for real-time or streaming audio/video transfer between browsers.

From the beginning of our project, we have focused on a second component of WebRTC, namely the Data Channel. The possibilities of leveraging WebRTC to change the way users share and access content on the Web make up our research focus for which not much related work exists today. The latest commercial efforts in this direction focus on different use cases such as site-supporting Browser-based CDNs[1,2] and file-sharing applications[3,4].

[1] https://peercdn.com/
[2] http://swarmcdn.com/
[3] https://rtccopy.com/
[4] https://www.sharefest.me/

# 3 Concept of a Browser-based P2P System

The objectives stated below and shall give a broad overview of the planned concept, while the latter sections dig deeper into the concept:

- The implementation shall allow browsers to form a P2P system.
- The resulting implementation shall be a JavaScript library that is usable as a simple drop-in by web applications. This library is then able to introduce a data sharing facility on top of the P2P system to all users of those web applications.
- The library shall use only standard browser techniques so that every user is able to benefit from it out of the box. The communication shall occur via WebRTC Data Channels, further explained below.
- The server component shall incorporate as little functionality as possible. In this way, we push most of the functions to the client-side application code and the server code becomes easily interchangeable, reflecting a mostly server-less architecture.
- There shall be an emulator that is able to run our code for testing and measurement purposes.

## 3.1 Functional Description

After defining our objectives, we were able to break down the functionality of the implementation into four building blocks, namely the *WebRTC Handshake Procedure*, *Connecting to a Bootstrap Node*, *Joining the P2P Network* and *Exchanging/Routing Messages* that are described here.

### WebRTC Handshake Procedure

Exchanging messages as well as forming and joining the P2P system involves making use of the WebRTC offer/answer handshake mechanism. Because it dictates some of the design decisions of the software architecture, we took great care in evaluating this procedure.

Connecting two peers using the WebRTC handshake involves a rather complex negotiation sequence [11] as shown in figure 3.1. The *sendOffer* and *sendAnswer* messages contain SDP-information that are exchanged using an arbitrary channel like a WebSocket connection to a signaling server. This allows the peers to learn about each other (e.g., NAT traversal options, supported codecs) and agree on parameters for the connection.

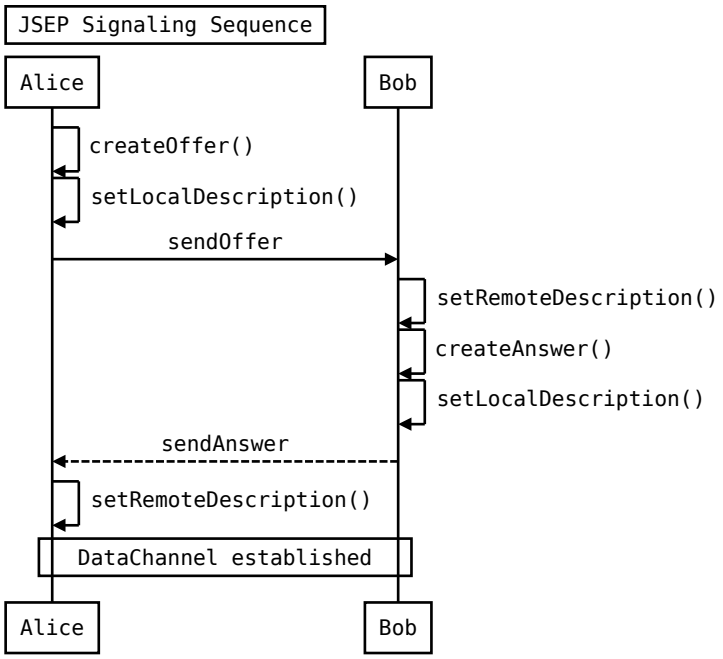


Figure 3.1: Alice establishes a WebRTC connection to Bob by following the JSEP signaling sequence using an arbitrary channel to transmit the offer/answer messages.

### Connecting to a Bootstrap Node

New peers join the P2P network by establishing a WebSocket connection to a bootstrap server that is connected to a certain number of peers which have previously joined the network. Thereafter this WebSocket connection is used to signal the WebRTC handshake, resulting in a direct WebRTC connection between the newly joined peer and at least one of the other peers. Once a peer has joined the network it may disconnect from the server without losing the capability of transferring data to/from other peers as shown in figure 3.2.

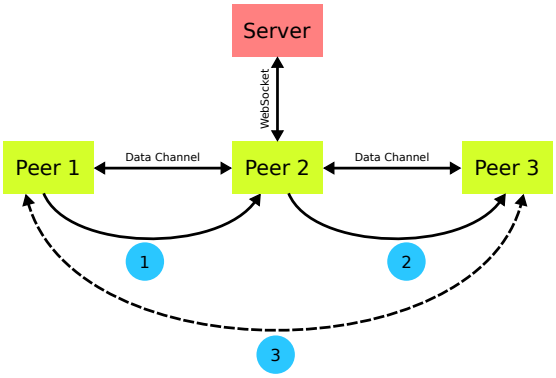


Figure 3.2: Peer 1 wants to communicate with peer 3. Therefore it sends the data via the open Data Channel to peer 2 which in turn routes it to peer 3. The server is not involved in this process at all.

### Joining the P2P System

After the initial bootstrap, the client has to discover other peers it shall connect to. This behavior depends on the type of the P2P network that is to be formed. A simple approach is to let the peer connect to all other known peers, eventually forming a full mesh. As this approach does not scale well with an increasing number of peers, a more sophisticated approach would be to form a structured P2P network using a DHT protocol like Chord [12] that guarantees logarithmic scaling to a large number of peers.

### Exchanging/Routing Messages

Message routing depends heavily on the chosen P2P network topology. Routing in a full mesh is trivial because the peers can directly exchange messages while a DHT imposes hop-by-hop routing as shown in figure 3.2. The figure demonstrates the exchange of a message between peer 1 and peer 3 via an intermediate hop (peer 2). Due to the peculiarities of the underlying WebRTC

technology, peer 3 cannot directly answer (because no Data Channel connection is established yet). Instead, it has to follow the WebRTC handshake procedure in advance (described above) or again use hop-by-hop routing to deliver the answer to peer 1.

## 3.2 Software Architecture

### 3.2.1 Component Oriented vs. Layered Approach

When outlining the components of our library and their interaction with one another, we explored several approaches previously proposed that dealt with the implementation of P2P overlay networks. The first one is proposed by Dabek et al. in [13]. The authors put forward a unified API for the implementation of structured P2P networks. As a part of this work, they analyzed common patterns of different structured P2P systems and abstracted them so that every overlay implementation can expose the suggested API without losing capabilities. This common API is called the key-based routing API (KBR). On top of the KBR layer, Dabek et al. identified additional abstractions that are only marginally elaborated on in the given paper.

The main idea behind the KBR API (or Dabek API) is that every structured overlay maps IDs from an ID space to every node employing a function specific to the implementation (Chord, Pastry etc.). This abstraction is then used to define KBR-specific API calls such as `route()`, `forward()` and `deliver()` for passing messages between nodes. Additionally the Dabek API defines methods for accessing the routing state on a node. In their evaluation the authors suggest implementation schemes for different applications on top of the KBR API. These include DHTs, group communication applications, and data replication mechanisms.

The proposed approach of exposing a handful of methods to establish a key-based routing and thus abstract away the actual routing implementation is flattering. Nevertheless we found the paper to be not extensive enough when it comes to the question of how the KBR layer is to be implemented. Especially in the context of WebRTC connection establishment and maintenance require a great deal of effort because of the offer/answer nature of JSEP. The authors of [13] do not mention how applications shall be able to bootstrap a local P2P node by connecting to another peer or a bootstrap server. Also they make no recommendation of how the KBR implementation shall interact with the underlying network, be it IP or – as in our case – WebRTC DataChannels.

Therefore we tried to find prior work that dealt with the actual details of implementing P2P systems and structuring code besides the API. Eventually we discovered OverArch, where the authors propose a detailed architecture for structured and unstructured overlay networks [14]. They start by pointing out the shortcomings of the Dabek API:

- It is focused on structured overlay networks thus disregarding unstructured P2P systems
- It provides no clear definition of neighbors, replica sets and r-roots
- The paper details only the KBR API, leaving out the other needed components in a P2P implementation
- The strict layered approach is too inflexible

Instead of just defining a set of APIs, the OverArch authors have fleshed out a detailed description of the required components of a P2P application implementation. These include a component for underlay and overlay connection management, a bootstrapping component as well as the services known from the Dabek API such as KBR, DHT and application-layer multicast (ALM). Each component encapsulates a certain functionality and exposes an API for leveraging this functionality to every other component.

This division into modular building blocks rather than strict layers makes it easier to orchestrate the components in different scenarios while maintaining exchangeability via a common API like Dabek et al. suggested. The authors mention the possibility of reusing one instance of a component in different applications. In our scenario of WebRTC connections this helps in that an application is able to provide the KBR layer with a custom bootstrap component (e.g. using a WebSocket connection to a dedicated server). Also an application could choose from a specific routing implementation. On top of that, OverArch specifies even the inner workings of the KBR module which helped us putting our implementation to work.

### 3.2.2 The BOPlish Architecture

The design of our architecture is presented in figure 3.3 and resembles the architecture proposed in [14]. At the very top sits the BOPlish API which is the entry point for all applications that make use of a P2P distributed content sharing facility. This is the developer facing part that exposes a set of simple methods for sending and receiving data. Below that part, we encapsulated a Router, a Connection Manager and a Bootstrapping mechanism.

The Router component is responsible for deciding where to forward packets to and thus maintains a routing table. Our current implementation uses a topology component that builds a full mesh of all peers for simplicity's sake. This component exposes those methods that are exposed by the KBR layer in the Dabek and OverArch API.

The Connection Manager component is responsible for handling WebRTC specifics like creating offers and answers, keeping track of open connections and handling glare, the latter is further explained in 4.1.3. To be able to join a P2P network, a node has to know at least one other

node already part of that network. This is where the Bootstrap component comes into play. It encapsulates the functionality for discovering an initial node to connect to. Since this process is very tightly bound to the generic connection establishment in our WebRTC-based implementation, we included this component into the Connection Manager.

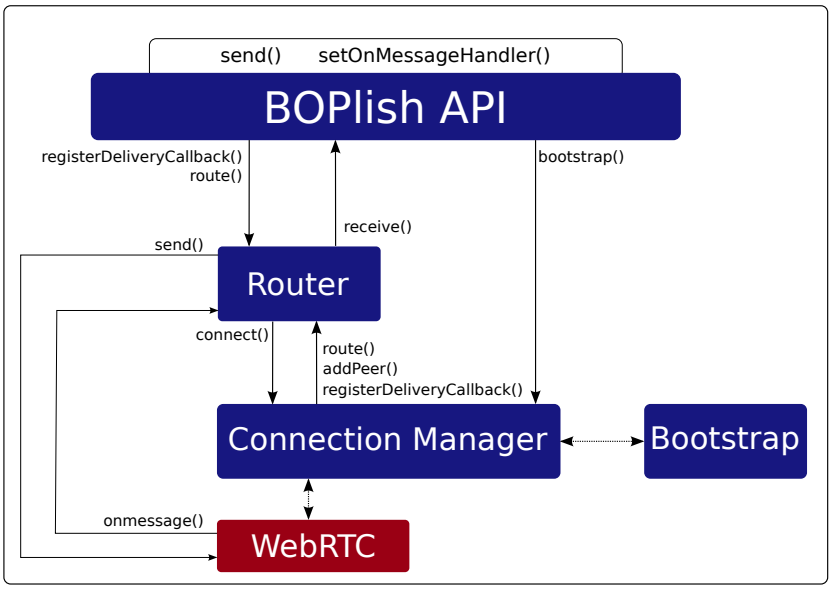


Figure 3.3: Outline of all BOPlish components and their interaction with one another. Each component is strictly defined by a set of API calls and loosely coupled to the other components. This enables us to easily replace parts of the functionality without having to refactor reliant code.

## 3.3 Security

The security of the application (server and clients) has not been a primary goal for our first iteration. There is no authentication built into our architecture so that peers cannot mutually identify themselves and everyone knowing and having access to the deployed application's URL can join the P2P user network. Communication security (and thus confidentiality and message integrity) through encryption is provided by the WebRTC implementations where every DataChannel is established on top of a Datagram Transport Layer Security (DTLS) connection [15]. Communication security between peers and the Bootstrap Server can be achieved by only serving WebSockets through TLS as well using the 'wss' URI scheme as defined in [16].

Since IDs are currently self-assigned and not authenticated in any way, malicious peers could join the network with a peer ID that already exists and probably disconnect other members. A secure ID generation algorithm with verifiable IDs must be integrated to mitigate this risk.

The WebRTC specification envisages an identity mechanism so that peers may mutually identify themselves using a third-party identity provider (IdP, [15]). This mechanism is currently not implemented in any browser but at least Mozilla is working on an implementation. This mechanism, though, is only specified for audio and video data passed between peers. Thus we are working on a concept of providing actual authentication and mutual identity verification without the need to inject a central server into our architecture. We have been reluctant to mandating any server-side authentication in our architecture because this would add complexity on the server-side, a fact that we wanted to avoid as much as possible in our vision of a server-less web.

Currently, the most reasonable approach is to employ public key cryptography. The W3C is working on standardizing cryptography functions exposed by browser implementations [17] so that web applications will be able to generate keys as well as sign, encrypt and verify data. As long as browser vendors have not implemented these features, we will make use of convenient JavaScript implementations.

The general security problems of P2P systems also exist with WebRTC as underlying transport. The DHT implementation must actively deal with malicious nodes, counter sybil attacks and force some sort of trust or reputation system for nodes and ensure the usage of secure identifiers. An overview of security issues and solutions in P2P systems is given in [18].

# 4 Implementation

## 4.1 Core BOPlish Library

We now want to explain the BOPlish library. As such, the different components of the implementation are now elaborated on in detail. At first, the BOPlish Client API and an example protocol are introduced before digging into the lower-level `Connection Manager` and `Router` components as well as supplementary resources such as unit tests. Figure 4.1 outlines the directory structure of the resulting project tree.

### 4.1.1 Client Instantiation

To establish a connection to a BOPlish User Network, the BOPlish client application has to be instantiated:

```
1 var bopclient = new BOPlishClient(bootstrapHostname, \
2   successCallback, errorCallback);
```

The *boostrapHostname* denotes the location of the bootstrap server while the *successCallback* respectively the *errorCallback* is used to indicate to the application whether the connection attempt has been successful or not. The BOPlish constructor then does the following:

- assigns a random `id` (SHA-1 hash) to this peer
- opens a WebSocket connection to the passed bootstrap servers hostname
- instantiates a `ConnectionManager`
- instantiates a `Router` and assigns the `ConnectionManager`

After the success callback has been called, this peer is connected to a minimum of one other peer in the system using the BOPlish bootstrap sequence. The `BOPlishClient` instance provides the `setOnMessageHandler()`-method to register application specific protocols and the `send()`-method to send messages using a predefined protocol as further described in 4.1.2.

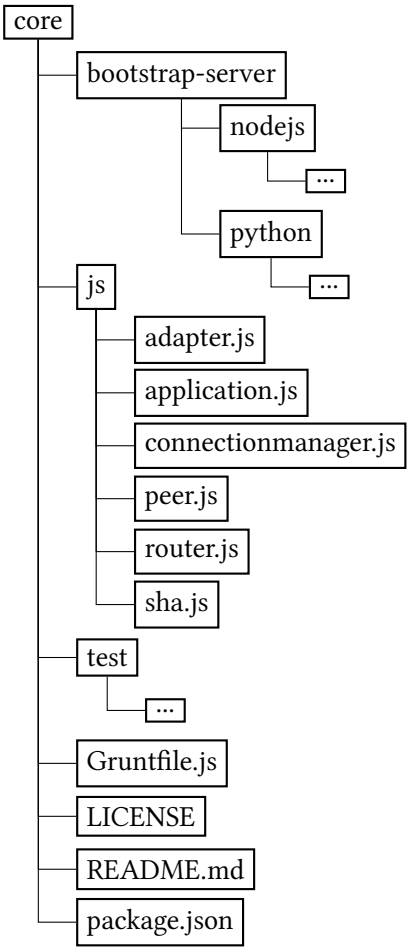


Figure 4.1: Directory tree of the core library project. The two proof-of-concept servers – written in Node.js and Python, respectively – reside in `bootstrap-server`, the library implementation code sits in `js` and `test` contains all unit tests. The root directory holds boilerplate files such as the grunt configuration, the license file, a Readme and a project file.

### 4.1.2 Client API & Protocols

A developer leveraging a BOPlish User Network has to implement its own application specific protocol on top of the core library to communicate with other peers. In this section, a simple request/response ping-protocol is elaborated on in detail to show how this procedure works. As described above, the client API consists of only two main functions, namely `send()` and `setOnMessageHandler()`.

A client protocol has to have a distinct name with which it is identified in the `Routing`-Component. In this example, the protocol is called ping-protocol. The Application Developer registers the ping-protocol in the BOPlish core by predefining a callback function and handling it over to the `setOnMessageHandler()` as follows:

```
var pingOnMessageHandler = function(msg, from) {
  if (msg.type === 'ping') {
    bopclient.send(from, 'ping-protocol', {type: 'pong'});
  } else if (msg.type === 'pong') {
    // calculate round trip delay
  }
};
bopclient.setOnMessageHandler('ping-protocol', \
  pingOnMessageHandler.bind(this));
```

This code registers the ping-protocol callback in the `Router` which can then identify incoming messages by the protocol name and call the corresponding callback. The protocol defines the subtypes *ping* and *pong* by setting the type-attribute accordingly. The Routing component is agnostic over these subtypes and only uses the distinct name of the protocol to identify message. This allows for extendable, yet simple-to-use client protocols.

Whenever the peer receives a `ping`-message, the `pingOnMessageHandler` is called by the `Router` and (in this case) answers with a `pong`-message by calling the `send()`-method on the `BOPlishclient`-instance with the recipient id, the protocol type and the payload. The ping-protocol is started by sending a `ping`-message to the remote peer that is to be pinged:

```
bopclient.send(toPeerId, 'ping-protocol', {type: 'ping'});
```

The `ping-protocol` described above is a simplified version of the implementation found in the demo repository (see *protocols.ping.js*). Other protocols used by either the demo applications described in 5.2 or the core library described below:

### Topology Protocol

The topology protocol is used to query neighboring peers for topology information (i.e. the IDs of their neighbors). To start the gathering process, the method `sendRequest()` is called which sends a `request` message to all peers in this peer's routing table. Upon receiving a `topo-protocol`-request, the protocol answers with a list of peers it is connected to.

When receiving the answer containing the connected peers, the application can decide whether to traverse deeper and send requests to the IDs of the peers contained in the answer. Contrary to a flooding-based approach, this behavior prevents the requesting peer from being overwhelmed by answers as the application can always decide to stop sending new requests.

The protocol is registered in the `Router` component (4.1.4) using the identifier `topo-protocol`. It defines the subtypes `request` and `response`.

### Discovery Protocol

The discovery protocol is used by the `Router` after the bootstrap sequence to find other peers in the network to connect to. This protocol is comparable to the topology protocol but simpler. As it is part of the bootstrap sequence, it is mandatory to implement. Rather than residing in its own file, it is defined in the core and built into the `Router` component.

To kick of the discovery mechanism, a `discovery-request` is sent to a remote peer which answers with a list of ids it is connected to. Upon receiving the answer, it is passed to the `ConnectionManager` component to proceed with the bootstrap sequence (see 4.1.3).

The protocol is registered in the `Router` component (4.1.4) using the identifier `discovery-protocol`. It carries the subtypes `request` and `answer`.

### Signaling Protocol

The signaling protocol is used to communicate with the bootstrap server over WebSockets and for in-band signaling over the peer-to-peer Data Channels. Like the discovery-protocol, the protocol is defined in the core (built into the `ConnectionManager`) as it is mandatory to implement. It encapsulates the WebRTC offer/answer mechanism and aggregates the different messages types in a BOPlish compatible format as further described in 4.1.3.

The protocol is registered in the `Router` component (4.1.4) using the identifier `signaling-protocol`. It carries the subtypes `offer`, `answer` and `denied`.

### Bopcast Protocol

The bopast protocol is a communication protocol that handles group management. Using this protocol, a simple application-layer multicast can be implemented. Figure 4.2 shows a sequence diagram outlining the steps to form a group of three participants. First up, Bob sends a `register-request` to Alice. She adds Bob to her list of receivers and acknowledges with a `register-response`. When Bob receives the response, he also adds Alice to his list of receivers. At this point, Carol wants to join the group. She sends a `register-request` to one of the peers in the group (Bob, in this case). Bob adds Carol to his list of receivers, acknowledges the request and propagates it to all existing members of the group (Alice, in this case). Alice receives the propagated request and also adds Carol to the group. Alice then sends an acknowledge to Carol which makes Carol add Alice too.

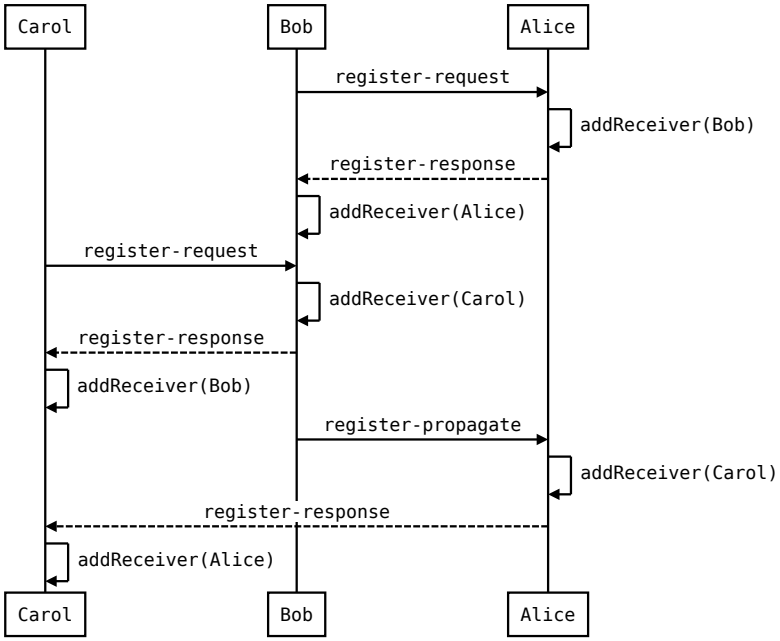


Figure 4.2: Bopcast sequence diagram showing the process of forming a group of peers

Bob, Alice and Carol can now communicate over the bopcast protocol using the `send`-method which delivers the message to every peer in the list of receivers.

The protocol is registered in the `Router` component (4.1.4) using the identifier `bopcast-protocol`. It carries the subtypes `register-request`, `register-propagate`, `register-response` and `deliver`.

### 4.1.3 Connection Manager

The Connection Manager (residing in `connectionmanager.js`) is responsible for establishing WebRTC Data Channels between peers. Every peer holds exactly one instance of the `ConnectionManager` class. To establish a connection to the P2P network, this class exposes the `bootstrap()` method that takes a `Router` instance as argument, creates an offer and lets the router forward this offer to another peer (see section 4.1.4 for details of how messages are forwarded). Upon receiving an appropriate answer, the Data Channel is established, a `Peer` instance created and passed on to the router for further processing (e.g. adding it to the peer table, depending on the routing protocol).

Since all connection establishment in WebRTC is asynchronous, the Connection Manager must store the state of every connection and act appropriately on every possible state change. Therefore we implemented the Connection Manager itself as a state machine that is outlined in figure 4.3. In fact, this state machine is divided into a "bootstrapping" phase and a "regular" phase.

As long as the Connection Manager resides in the bootstrapping phase, it does not accept incoming connection requests until a first connection is established to reduce complexity. This can be interpreted as the "happy path": Create an offer, wait for the answer, accept the answer and pass on into the regular phase. In the regular phase the Connection Manager can be used to actively initiate a connection (create offer, send offer to other peer, receive answer, accept answer, wait for connection establishment) or accept incoming connection requests (accept offer, create answer, send answer to peer, wait for connection establishment).

#### Glare Handling

Between all these states, though, scenarios different from the happy path may occur, e.g. after creating and sending an offer the Connection Manager may receive an offer from the exact same peer it has sent an offer to. In this situation we have two competing offers that – not properly handled – would result in two Data Channels between two peers. This situation is called glare or call collision and must be handled by WebRTC applications.

Our simple approach to handling this situation relies on the uniqueness of the "sess-id" field in the "o=" line of every initial offer. The current draft version of [11] states:

> The value of the <sess-id> field SHOULD be a cryptographically random number. To ensure uniqueness, this number SHOULD be at least 64 bits long."

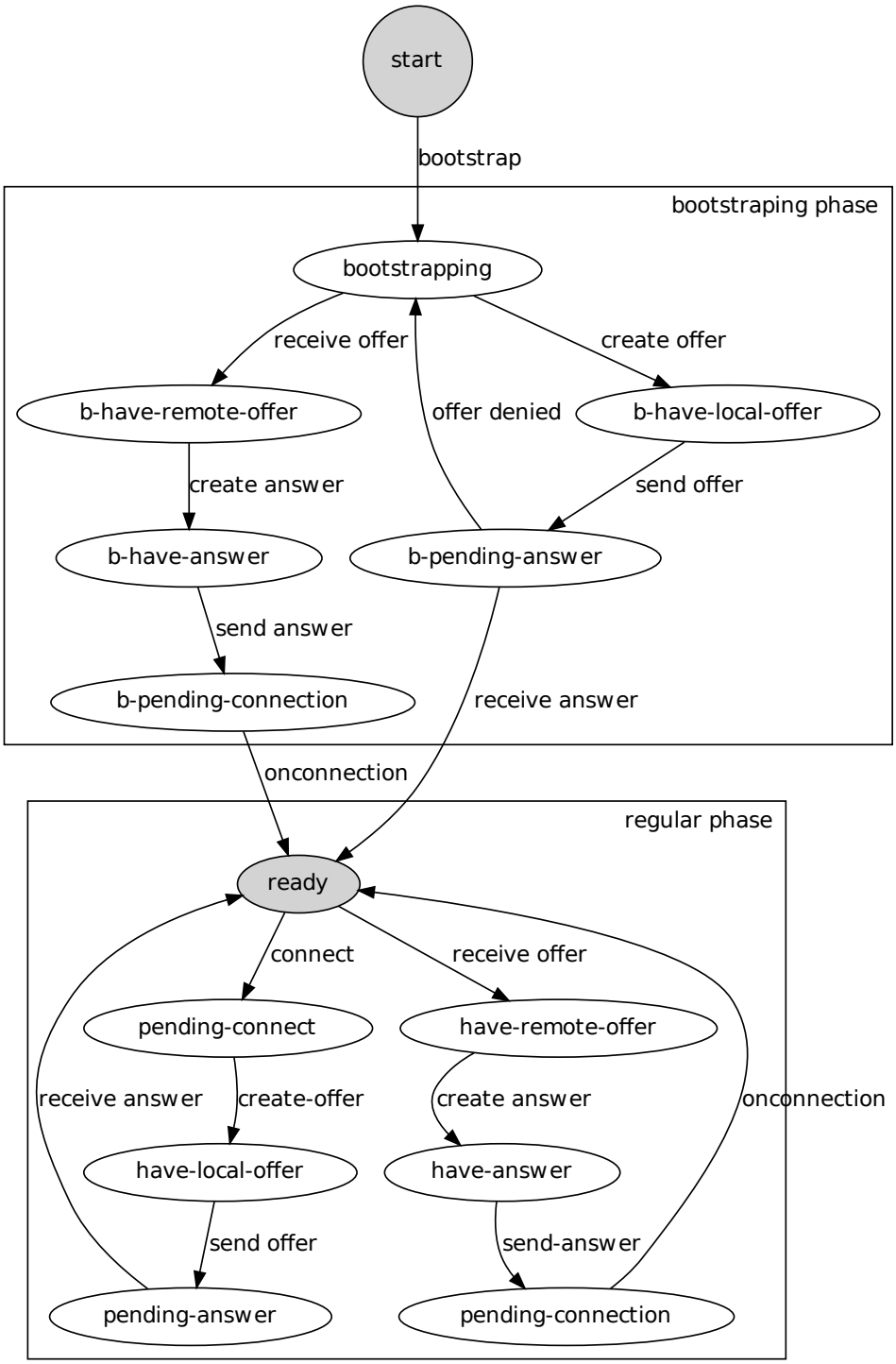


Figure 4.3: The Connection Manager implements a state machine consisting of a bootstrapping and a regular phase. The bootstrapping phase is entered for acquiring an initial connection to an arbitrary peer. After this connection has been established, the regular phase is entered in which additional connections can be established.

Although the draft only puts a SHOULD constraint on the concrete field value we have experienced that at least Firefox and Chrome use (non-cryptographically) random unique values[1].

When an offer A is received in the regular phase by the Connection Manager, it checks whether an offer has been sent to the sender of A. If this is the case, the session IDs of both offers are compared and the offer with the lower session ID is discarded. This way both peers synchronize through the session ID, know which offer is discarded and can react appropriately without any additional message exchange.

In the bootstrap phase, though, the Connection Manager does not know where the first offer is sent to because the recipient is chosen by the bootstrap server. Therefore every offer that is received while the peer is bootstrapping is regarded as coming from a potential bootstrap partner and the same glare handling as in the regular phase is applied.

### 4.1.4 Router

The Router component constitutes the heart of the core BOPlish library. It is responsible for disseminating messages arriving from the client layer through the network. Additionally it receives messages from the bootstrap server and maintains its routing table. These four methods make up the public API of the Router:

- `addPeer(peer)`
- `getPeerIds()`
- `route(to, type, payload)`
- `registerDeliveryCallback(msgType, callback)`

Since the other components don't make any further assumptions about the Router it can be exchanged quickly and easily. Our current implementation creates a fully meshed P2P network, in the next iteration we plan to exchange this implementation with a Chord DHT. The following description focuses on the full mesh implementation.

All messages that enter and leave the Router component are simple JSON objects that have the properties `type`, `from`, `to` and `payload`. The `payload` designates the application-layer data that is delivered to the registered callback. An example routing-layer message looks like this:

```
1 {
2   to: "f9c89ceb726436bb0d7074c08788d08e0e974dbf",
```

[1] https://mxr.mozilla.org/mozilla-central/source/media/webrtc/signaling/src/sipcc/core/gsm/gsm_sdp.c#5434

```
  from: "48142a86bd1dc7c2be81df1c5ca6d0a98c328f4b",
  type: "ping-protocol",
  payload: {
    "date": "Mon Nov 11 2013 17:44:05 GMT+0100 (CET)",
    "type": "pong"
  }
}
```

When a Router instance receives a message, it first checks whether the `to` field equals the ID of this instance. If this is the case, the registered callback (if any) is invoked with the `payload` and `from` fields as parameters. If the `to` field is not equal to the Router instance's ID the Router searches for a peer with that ID in its peer table. If it finds one, the message is passed as is via the existing DataChannel connection. Otherwise, the Router forwards the message through the fallback signaling channel which is a WebSocket connection to the signaling server in our case. Since the server holds connections to all connected peers it knows exactly where to forward the message to.

As stated in section 4.1.3 the Connection Manager uses the Router to even route the first offer to an arbitrary peer. Since the remote peer's ID is unknown in this state, the `to` field is left empty which causes the Router to forward the message through the Bootstrap Server. The server then designates a connected peer as the bootstrap peer and forwards the offer through the corresponding WebSocket (for details about how this is done see section 4.2).

After the first peer has been added to the peer table via the `addPeer` method, the full mesh Router starts a neighbor discovery by sending a message of the type 'discovery-request' to its only peer. The latter then sends a 'discovery-answer' with a list of all its connected peer's IDs as payload. Upon reception of this answer the Router mandates the Connection Manager to connect to each of these peers eventually connecting to all peers in the network.

### Limits

The described mechanism for bootstrapping the P2P network using discovery messages is rudimentary and may result in an incomplete mesh configuration as seen in figure 4.4. On top of that a full mesh construction using WebRTC peers does not scale well to the extent that browsers have hard-coded limits of the maximum number of calls. For current Firefox implementations this limit is set to 51[2].

[2] https://mxr.mozilla.org/mozilla-central/source/media/webrtc/signaling/src/sipcc/core/includes/phone_platform_constants.h#193

Since our plan is to replace this Router with a DHT implementation we will not handle these limits any further and instead focus on developing the DHT.

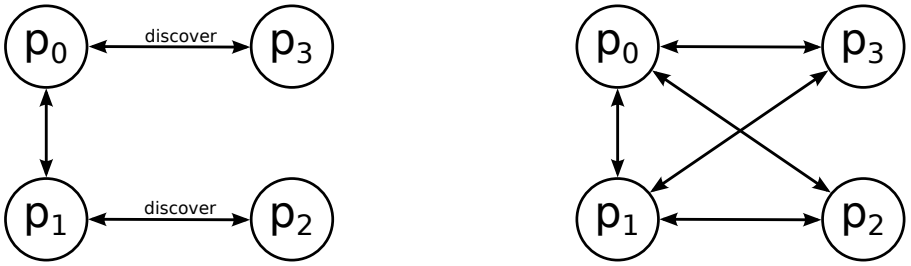


Figure 4.4: When two peers $p_2$ and $p_3$ join the network in parallel – each using a different bootstrap peer – as seen in (a) the current Router implementation may lead to a network as outlined in (b) where no full mesh is constructed because the link between $p_2$ and $p_3$ is missing.

## 4.2 Bootstrap Server

The Bootstrap Server has two distinct tasks: Deliver the application and act as fallback signaling channel using WebSockets. For delivering the application, no special functionality must be implemented because the HTML, JavaScript and CSS files are simply statically delivered to clients. For fallback signaling the server must handle WebSocket connections to URLs of the form '/ws/*PEERID*' where '*PEERID*' denotes the Peer's self-assigned ID. Offers and answers must be forwarded via the appropriate WebSocket connection using the 'to' field from the routing message. Initial offers that contain an empty 'to' field must be forwarded to a random peer that is different from the one in the 'from' field. The specified message format and behavior for signaling allowed us to implement two independent servers using Python and JavaScript, respectively, which are described further below.

### 4.2.1 Python

The Python implementation using the Flask web micro framework[3] is very simple and consists of 80 lines of code. For bootstrapping the environment the well-established tools virtualenv and pip are used that ship with most Linux distributions and are available for MacOS X, too. Using these tools all necessary libraries are installed to the local environment with the command

[3] http://flask.pocoo.org

`pip install -r requirements.txt`. Starting the server is as simple as calling `python run.py` which causes the server to listen on TCP port 5000 at 0.0.0.0.

### 4.2.2 Node.js

Node.js is a server-side runtime environment for JavaScript applications. It offers a built-in web server that is used to deliver the static files to the browser along with a server-side WebSocket implementation to handle the fallback signaling traffic. npm, the standard Node.js packet manager is used to install the dependencies via a single `npm install` command in the bootstrap server's path. Starting the server is done similar to the python implementation by calling `node run.js 0.0.0.0 5000`. As the test framework is also using Node.js, this module can be conveniently used for unit testing 4.3.

## 4.3 Environment

During the implementation phase of the BOPlish project, we have used several third party software to aid the development process.

**Source Code Management** We used a shared git repository for revisioning our code. The first iterations were stored on a private server and after releasing the code on GitHub[4] we now work on a public GitHub repository all the time.

**Build Process** For automating tasks such as generating HTML documentation from the annotated source code, running unit tests or building a minified version of the JavaScript library we opted for the Grunt JavaScript task runner[5]. A single file named Gruntfile.js serves as configuration for the different tasks. This makes it very convenient to e.g. run tests where one just calls `grunt test`.

**Testing Strategy** Since the very beginning of the BOPlish project, we created unit-tests alongside with the code that helps to reach a certain level of code quality. We opted for the mocha testing framework[6] which is running on Node.js. Additionally, we are using the sinon.js[7] framework that allows for stubs and mocks in an asynchronous context. We also created a Grunt task to run the tests during the build process.

[4] https://github.com/boplish
[5] http://gruntjs.com/
[6] http://visionmedia.github.io/mocha/
[7] http://http://sinonjs.org/

**Documentation Strategy** We started to document our code from the very beginning using JSDoc[8]. This works similar to the well-known Javadoc documentation format where comments in the source code are augmented with annotations so that classes, methods, members and callbacks are recognized as such. A Grunt task integrates JSDoc into our build environment so that `grunt jsdoc` builds the documentation.

**Deployment Strategy** Currently, we are deploying BOPlish by updating the code repository and using the build process on the deployment target server manually. When the project is more mature, we are planning to automate this task.

[8] http://usejsdoc.org/

# 5 Applications

## 5.1 Emulator

The efforts of implementing the WebRTC protocol stack are currently mainly focused on browsers. As of today, Google Chrome and Mozilla Firefox have the most complete implementations. An initial goal of the project was to implement an emulation component that allows for unit and scalability testing. Because using a complete browser for such a task is inefficient, we planned for a solution that works on the command line and can easily be scripted. This becomes especially important when the number of peers in the network rises as manual starting and testing a growing number of nodes is a tedious task.

The initial plan was to use the WebRTC components of the browser on its own. Unlike Mozilla, Google encapsulates the WebRTC components in the libjingle library[1]. As this C++ library is not dependent on the browser instance, it can be run independently. By creating bindings between the C++ code and an adapter, an emulation component could be created. Using Node.js as the base for such an emulator would allow us to reuse the existing BOPlish code which is mainly written in JavaScript and, apart from the WebRTC components, could run in Node.js out of the box without requiring a complete browser stack.

While this solution seems to be an applicable task, multiple problems arose during the planning phase. First of, using the libjingle library means tying the emulator to the Google implementation. At the time of planning the emulation component, the SCTP stack that allows for SCTP-based Data Channels was not implemented yet but the Data Channels were instead based on an deprecated SRTP approach. Because of this, libjingle broke compatibility with the Mozilla Firefox implementation which was our main development platform. Secondly, the libjingle library was very much in flow, resulting in a buggy build process and constant API changes making it hard to adapt to the rapid updates.

As of today, the situation has improved a lot and the implementations will be compatible in the foreseeable future. Multiple projects aiming to bring WebRTC support to Node.js have since been started using the same approach that has been described above. The most promising

[1] https://code.google.com/p/webrtc/

ones are node-webrtc[2] and node-peerconnection[3]. As soon as the WebRTC implementation are interoperable, we will take a closer look at the libraries that yield the binding between libjingle and Node.js, eventually allowing us to run BOPlish without a browser. The next step would then be to build an statistic gathering as well as an evaluation component.

## 5.2 Demo Applications

During the work on the BOPlish core, we implemented several demo applications that assisted in showcasing our work. Moreover, the demos allow to prematurely find problems related to the user-facing BOPlish API (see figure 3.3 for an overview on the API). To separate the core BOPlish library from the application code, the demos reside in its own repository[4].

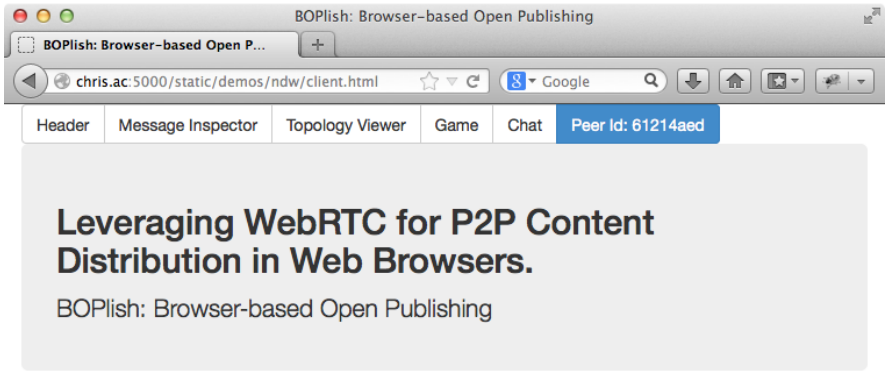


Figure 5.1: BOPlish demo client interface in a Firefox Browser

The demo applications use the Bootstrap CSS Framework[5] that allows for cross-browser web frontend development. Even though all major browsers are capable of displaying the frontend, only Firefox (stable build) and Chrome (nightly build) have currently implemented the mandatory WebRTC components to run BOPlish. The demo is reachable by pointing a user's browser to a HTTP URL where a standard web server (e.g. apache) serves the files to the browser. When loaded, the application shows a header and menu bar with entries for the various demo applications as well as a field that contains the randomly chosen id of this peer (see figure 5.1).

[2] https://github.com/modeswitch/node-webrtc
[3] https://github.com/Rantanen/node-peerconnection
[4] https://github.com/boplish/demos
[5] http://getbootstrap.com

Upon startup, the demo application establishes a WebSocket connection to a BOPlish bootstrap server and instantiates a boplish client that uses the `signaling-protocol` to connect to the network (see 4.1.2).

### 5.2.1 Message Inspector

The purpose of the Message Inspector app is to showcase the communication between the peer and the network and an simple, real-time debugging interface for the BOPlish protocols. The applications view (see figure 5.2) is separated into three colums. The first column shows a list of peers this peer is connected to (ids are shortened to 8 characters). The functionality of the two buttons next to the peer id is described below:

- *Ping*: sends a `ping-protocol` request to the corresponding peer (see 4.1.2)
- *Bopcast Registration*: sends a `bopcast-protocol` register-request to the corresponding peer (see 4.1.2)

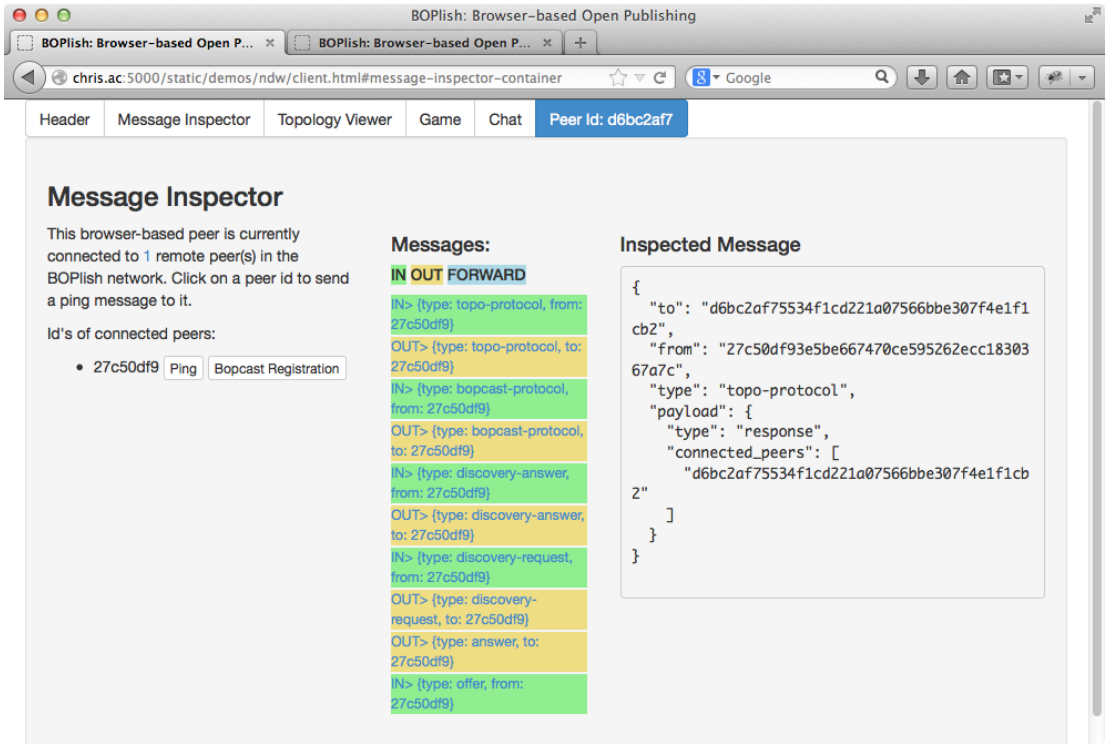


Figure 5.2: BOPlish Message Inspector demo application

The second column shows all messages that are handled by this peers `Router` instance. Incoming messages are marked green, outgoing messages are marked yellow. Blue markings indicate a message that is forwarded to another peer in the routing table as the receiver is not this peer. The messages are inserted on top of the list in real time. Upon clicking a message in the list, the third column shows the messages content.

### 5.2.2 Topology Viewer

This demo application showcases the topology of a BOPlish User Network by displaying a graph of the network (see figure 5.3). Nodes in the graph reflect the different peers while a dark blue colored node mirrors the peer that is running the topology viewer. Links between the nodes show a Data Channel connection between the corresponding peers.

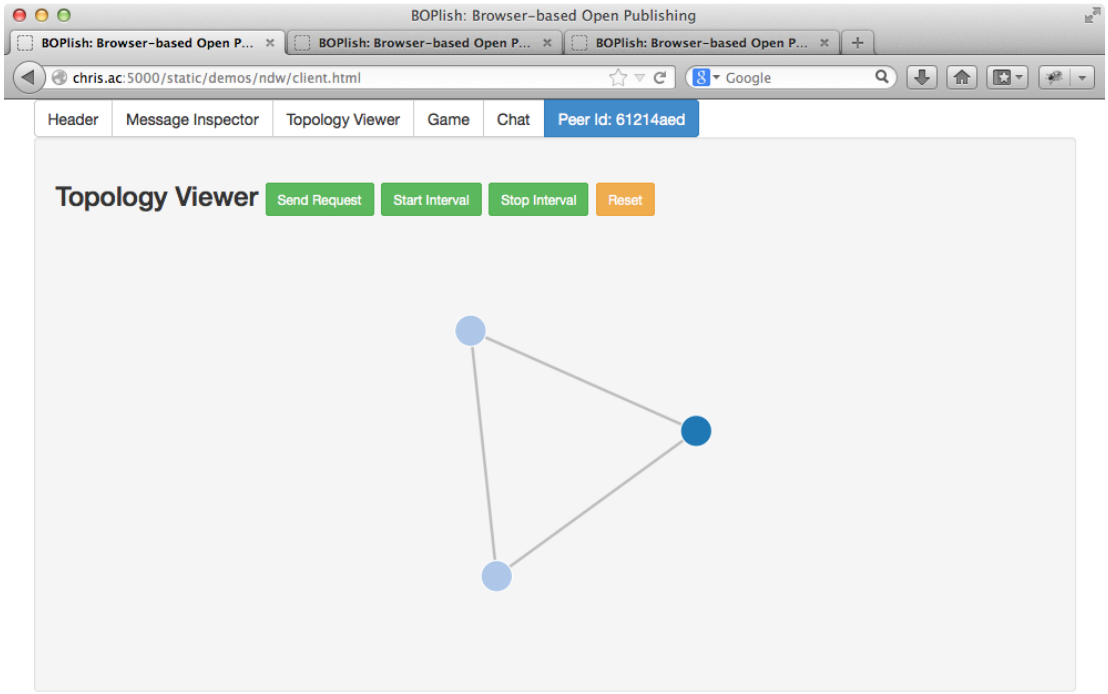


Figure 5.3: BOPlish Topology Viewer demo application

Buttons allow the user to interact with the `topo-protocol` that is used to gather the needed information (see 4.1.2). *Send Request* sends a single `topo-protcol` request to all connected peers. Upon reception of the answers, the response is fed into a module that renders the graph using

the d3.js library[6]. *Start* and *Stop Interval* can be used to start/stop a sequential dispatching of `topo-protocol` requests to automatically update the graph when new peers join the network. Finally, *Reset* resets the graph and makes the application forget all the node and link information learned.

### 5.2.3 Game

The game is a simple application that uses the bopcast-protocol to showcase multi-user group communication. After registering other peers using the `bopcast-protocol` procedure, all peers in the registered group can use the controls to move the red circle on the black grid. All changes to the position will be synchronized to the whole group.

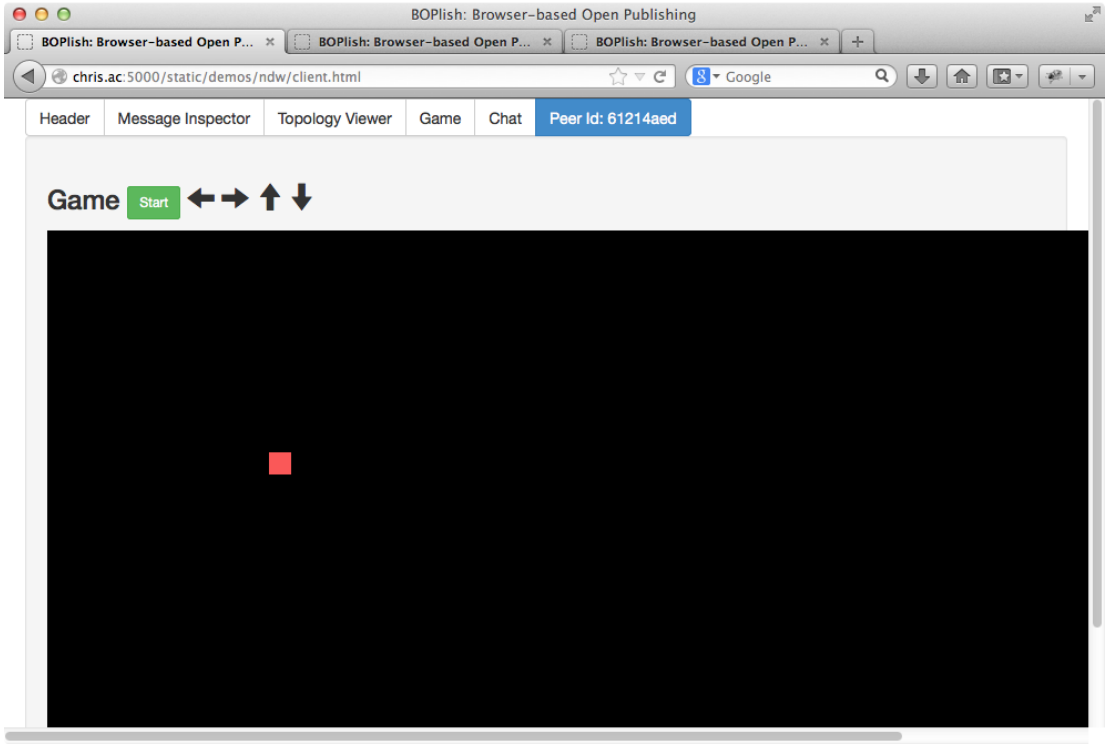


Figure 5.4: BOPlish Game demo application

[6] http://d3js.org/

### 5.2.4 Chat

Similar to the game application, the chat also uses the `bopcast-protocol` to send and receive data to/from groups of recipients. After registering a group, all participants can send and receive textual chat messages. A username can be freely chosen while it defaults to the id of the peer.

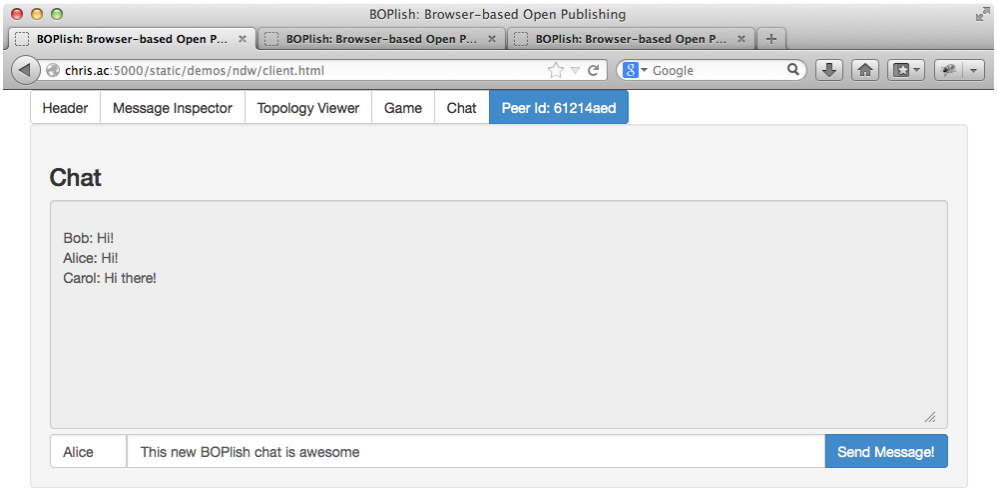


Figure 5.5: BOPlish Chat demo application

# 6 Conclusions and Outlook

In this report, we have introduced an extendable architecture that lays the foundation for a decentralized content-publishing facility between browsers. Our current experience of combining WebRTC with Peer-to-Peer networking leaves us confident that a deeper exploration of the possibilities provided is well worth it. We have already planned next steps with regards to further investigating user-centric content publishing in browsers by extending our current implementation [19]. We are intensely watching and participating in the ongoing research and implementation activities by W3C, IETF and browser vendors to handle potential problems/incompatibilities caused by the non-final state of the specifications as early as possible.

Topics that are to be investigated more thoroughly with regards to WebRTC are the security and privacy of users. We will further cover these as part of our ongoing research and implementation efforts, as well as refine the architecture described in this report and in [19] to have an implementation that enables interest groups to share content by name without relying on a central content server or having to register DNS names (see figure 6.1 for a broad overview). The next steps are divided into these categories:

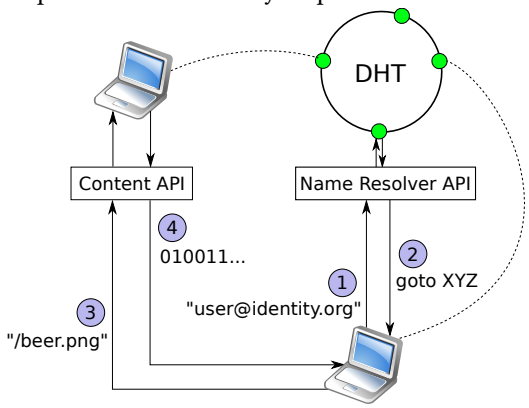


Figure 6.1: Our vision of a name-based content publishing architecture using WebRTC as further outlined in [19].

- Replace the full mesh Router with a Chord DHT implementation.
- Incorporate a standalone WebRTC emulator for functionality and performance testing.
- Investigate strategies to further simplify the BOPlish deployment.
- Implement the Naming and Content Retrieval API.
- Investigate security enhancements with regards to authentication, ID generation/assignment and privacy.

# Bibliography


[1] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "WebRTC 1.0: Real-time Communication Between Browsers," World Wide Web Consortium, W3C Working Draft.

[2] H. Alvestrand, "Overview: Real Time Protocols for Brower-based Applications," IETF, Internet-Draft – work in progress 01, June 2011.

[3] C. Vogt, M. J. Werner, and T. C. Schmidt, "Leveraging WebRTC for P2P Content Distribution in Web Browsers," in *21st IEEE Intern. Conf. on Network Protocols (ICNP 2013), Demo Session.* Piscataway, NJ, USA: IEEEPress, Oct. 2013, iCNP Best Demo Award.

[4] R. Jesup, S. Loreto, and M. Tuexen, "RTCWeb Data Channels," IETF, Internet-Draft – work in progress 04, February 2013.

[5] G. Xylomenos, C. Ververidis, V. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. Katsaros, and G. Polyzos, "A survey of information-centric networking research," vol. PP, no. 99, 2013, pp. 1–26.

[6] M. Wählisch, T. C. Schmidt, and M. Vahlenkamp, "Backscatter from the Data Plane – Threats to Stability and Security in Information-Centric Network Infrastructure," *Computer Networks*, vol. 57, no. 16, pp. 3192–3206, Nov. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2013.07.009

[7] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A Survey of Information-Centric Networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, July 2012.

[8] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram, "Maygh: building a cdn from client web browsers," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 281–294.

[9] A. J. Meyn, "Browser to Browser Media Streaming with HTML5," Master's thesis, Technical University of Denmark, DTU Informatics, E-mail: reception@imm.dtu.dk, Asmussens Alle, Building 305, DK-2800 Kgs. Lyngby, Denmark, 2012.

[10] A. Amirante, T. Castaldi, L. Miniero, and S. Romano, "On the seamless interaction between webRTC browsers and SIP-based conferencing systems," *Communications Magazine, IEEE*, vol. 51, no. 4, pp. 42–47, 2013.

[11] J. Uberti and C. Jennings, "Javascript Session Establishment Protocol," IETF, Internet-Draft – work in progress 05, October 2013.

[12] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[13] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003, Revised Papers*, ser. LNCS, M. F. Kaashoek and I. Stoica, Eds., vol. 2735. Berlin Heidelberg: Springer–Verlag, 2003, pp. 33–44.

[14] I. Baumgart, B. Heep, C. Hubsch, and A. Brocco, "OverArch: A common architecture for structured and unstructured overlay networks," in *2012 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2012, pp. 19–24.

[15] E. Rescorla, "WebRTC Security Architecture," IETF, Internet-Draft – work in progress 07, July 2013.

[16] I. Fette and A. Melnikov, "The WebSocket Protocol," IETF, RFC 6455, December 2011.

[17] D. Dahl and R. Sleevi, "Web Cryptography API," World Wide Web Consortium, W3C Working Draft.

[18] H. Schulzrinne, E. Marocco, and E. Ivov, "Security Issues and Solutions in Peer-to-Peer Systems for Realtime Communications," IETF, RFC 5765, February 2010.

[19] C. Vogt, M. J. Werner, and T. C. Schmidt, "Content-centric User Networks: WebRTC as a Path to Name-based Publishing," in *21st IEEE Intern. Conf. on Network Protocols (ICNP 2013), PhD Forum.* Piscataway, NJ, USA: IEEEPress, Oct. 2013.