OakStreaming: A Peer-to-Peer Video Streaming Library

István Koren and Ralf Klamma

Advanced Community Information Systems (ACIS) Group,
RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
E-mail: koren@dbis.rwth-aachen.de; klamma@dbis.rwth-aachen.de
http://dbis.rwth-aachen.de

Received October 2018;
Accepted February 2019

Abstract

Multimedia platforms dealing with movie streaming and video-based short messages have increased the global Internet video traffic substantially in the last couple of years. Over the same period, multimedia on the Web has been standardized in terms of codecs and browser-based JavaScript APIs. However, today the technological challenges concerning the distribution of large video files are mainly tackled by scaling up capacities in cloud data centers, or relying on content delivery networks. Both approaches favor financially strong, large companies, while independent video providers with highly demanded videos are disadvantaged. Peer-to-peer streaming provides an alternative by shifting the data streams to the clients. In this article, we conceptualize different methods to move video delivery from centralized cloud infrastructures to end user devices. We discuss their strengths & weaknesses and present design considerations. To exemplify a particular approach, we showcase the implementation and evaluation of OakStreaming, our system that streams videos peer-to-peer via WebTorrent in HTML5.

Journal of Web Engineering, Vol. 17_6&7, 527–560.
doi: 10.13052/jwe1540-9589.17675

Particularly, we offer Web video providers a library that has various parameters, for instance to limit the bandwidth available for peer-to-peer uploads. The resulting library is available as open source software on GitHub.

Keywords: Web Engineering, Video Streaming, Peer-to-Peer, WebRTC, WebTorrent.

1 Introduction

Global Internet video traffic has risen substantially in the last couple of years. The continuous increase of connection speed has paved the way for the success of multimedia-related businesses dealing with movie streaming and video-based short messages. In 2016, around 73% of all consumer Internet traffic regarding bits transferred was video traffic [3]. According to Cisco's *Visual Networking Index*, this number will most likely rise to 82% in 2021. Large Internet companies like *Facebook* and *Twitter* entered the video business, trying to gain a share of the consumer market dominated by *YouTube* [10]. There is also a movement towards live video on social networking sites. As a social phenomenon, *digital natives* prefer emergent and often short-lived social networks like ~~*Musically*~~*TikTok* and ~~*Snapchat*~~*Instagram* [*intentional typography*], whose business models include short video "stories" that vanish after a certain time period. Besides entertainment and social networking aspects, there is also huge potential in further multimedia-based application cases like distant, collaborative work-places or visual instructions using augmented reality. Users get more acquainted with local multi-device streaming of videos from mobile computers to the big screens of smart TVs due to new "second screen" options. Simultaneously, video bitrates increase steadily. 4K with a horizontal screen display resolution of around 4,000 pixels is now state-of-the-art in home entertainment, but 8K is already on the horizon with the recent HDMI specification [23].

Meanwhile, a shift of multimedia technologies on the Web from proprietary plugins like Adobe Flash to a standardized set of HTML5-related standards has taken place. Necessary preconditions for cross-device multimedia on the Web were tackled, including licensing

issues around media codecs such as WebM and MPEG H.264. However, technological challenges in distributing large video files today are mainly addressed by relying on central cloud providers with large storage capacity, so that the de facto video distribution model still follows the client-server architecture. Video files are first uploaded from client devices to a central cloud solution, before being downloaded or streamed to clients. This entails a number of drawbacks. First, live streaming takes a temporal indirection when routed over a server. Second, it generates a high load on central points of the network infrastructure. Last but not least, video creators must accept the terms and conditions of the cloud provider to serve the video. Alhough *content delivery networks (CDNs)* solve the first problem, they introduce a further monetary burden, making it hard for small content providers to reach the quality and availability of big providers. Thus, a possible solution lies in shifting the load *from* the cloud *back* to the clients, following general ideas of *edge computing* [20]. Peer-to-peer video streaming is a promising principle to meet these challenges. Although browser plugins like *Adobe Flash* and *Akamai Netsession* have accomplished this already several years go, they have not reached mainstream adoption due to usability and security shortcomings. Today, two recent group of Web standards are here to solve the challenges natively in the browser. On the one hand, the *Media Source Extensions* control feeding `<audio>` and `<video>` tags with multimedia data [25]. On the other hand, Web Real-Time Communication (WebRTC) as a group of protocols and APIs solve issues around cross-browser peer-to-peer data streaming [1].

Streaming video files needs to tackle a number of challenges like the following core requirements [7, 21]:

- Fast initial startup time: There should be no significant lag when starting a stream.
- Random access: It should be possible to switch the position in the timeline.
- Complying with device and network limitations: The resolution of the target device and the bandwidth need to be respected.

Although technologies and initial prototypes are now available, in our opinion the scientific literature lacks a comprehensive overview of

possible alternatives to server-centric video streaming systems and a discussion of their costs and benefits. In this paper, we contribute conceptual considerations and a prototype implementation of a framework for peer-to-peer video streaming in native HTML5. We elaborate on the conceptual architecture of our solution, which is relying on a tracking server. Concretely, we decided to use an approach using *WebTorrent*[1]. Amongst other parameters, developers may specify a limit on the amount of data each peer is allowed to upload to other peers, for keeping a balance amongst the peers. The resulting library called *OakStreaming* is available as configurable and extensible open source solution, enabling the further development with the help of the open source community.

This article is organized as follows. After introducing related Web technologies and academic work in Section 2, we present some functional and non-functional requirements in Section 3. Section 4 discusses different concepts for peer-to-peer multimedia streaming on the Web. The resulting OakStreaming library that enables setting up browser-based peer-to-peer video streams is presented in detail in Section 5 and evaluated in Section 6. Section 7 concludes the paper with a discussion and pointers to future work.

2 Related Work

This section introduces the technological background and standards of video streaming on the Web and derives design considerations for distributed approaches. We then present related work from academia and industry that tackles the challenges of delivering video content through peer-to-peer technologies.

2.1 Web Video Technologies

WebRTC refers to a set of HTML5 APIs that enable establishing direct peer-to-peer connections between two or more Web browsers [1]. The *data channel* is a communication channel type of WebRTC which enables any binary data to be exchanged between Web browsers. In

[1] https://webtorrent.io/

order to use the WebRTC data channel, a website does not need the explicit agreement from the visitor, however the Web application needs to be delivered over a secure HTTPS channel.

To establish a WebRTC connection, signaling data has to be exchanged between the peers [1]. This includes the public IP addresses of each peer and the local IP addresses, if the peers are located in the same LAN. The data is typically exchanged with the help of a signaling server that is reachable from both ends of the connection. When using this approach, both peers connect to the signaling server and use it as a relay for transferring their signaling data to the other peer. The signaling data can also be transferred by other means. For instance, the peer-to-peer WebRTC connection can be established by manually typing in signaling data into an HTML form. Because of firewalls and the functional principle of *network address translation* (NAT), direct IP-based communication between peers is normally not possible on the Web. Therefore, if any two peers are located in different local networks and at least one peer is behind a NAT-enabled router, a so-called STUN (Session Traversal Utilities for NAT) server is necessary to pass through the router so that the WebRTC connection can be established.

The HTML5 `<video>` element was originally introduced to allow playing videos without proprietary plugins. It has one or more `<source>` children that specify media resources represented by URLs. However, additional measures are required for further client-side control of media streams in JavaScript. The W3C Media Source Extensions specification enables JavaScript to send byte streams to media codecs [25]. Web browsers that support this specification enable JavaScript code to "feed" a HTML5 `<video>` or `<audio>` element with a video/audio stream piecewise. It enables the implementation of client-side prefetching, buffering and adaptive bit rate for streaming media in JavaScript.

2.2 Client-Server vs. Content Delivery Networks

High server load and world-wide latency challenges are usually tackled with Content Delivery Networks (CDNs). Figure 1 compares client-server and CDN architectures. A CDN consists of a world-wide network

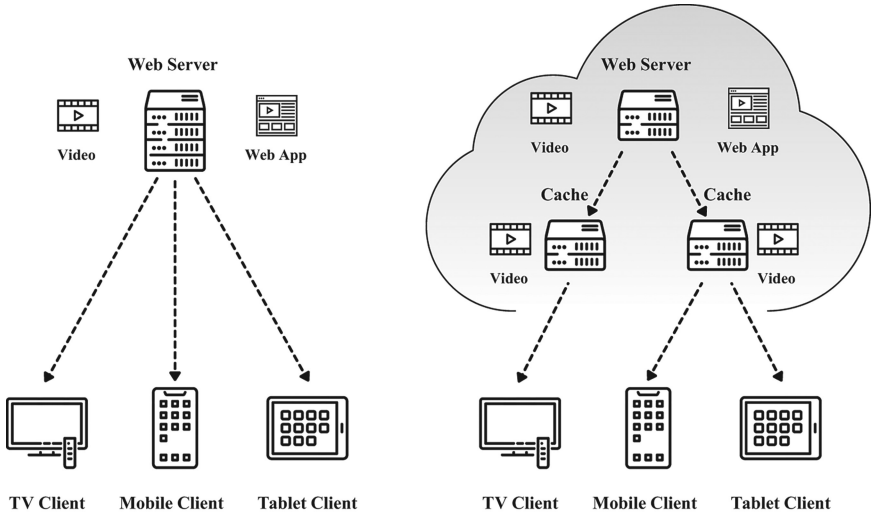


Figure 1 Comparison of conventional video streaming architectures.

of caches that replicate content of a main server. Requests to the central identity are then redirected to the nearest cache [2]. A commercial CDN provider usually delivers content for many different content providers. For the Web user, CDNs are transparent; the content provider, however, usually needs to pay fees.

Coolstreaming and *Akamai NetSession* are two representatives of systems that relay the caching to the peers. Lin et al. described and evaluated the peer-assisted CDN Akamai NetSession [15]. The peers of the network need to run instances of the NetSession Interface application. By including a library into third-party applications like download managers, they can also benefit from the network. Running as part of a download manager, a NetSession Interface instance first tries to download the fragments of a file from other peers of the NetSession network. Fragments that cannot be received fast enough are downloaded from dedicated servers of the CDN. Additionally, a running NetSession Interface uploads already downloaded fragments to other NetSession Interface instances. For both systems to work, a client-side software needs to be installed on the local operating system, possibly adding security vulnerabilities. Depending on the number of

devices employed by a user, different platform-specific versions must be manually downloaded and configured.

2.3 WebRTC-Based Prototypes

Högqvist et al. describe and evaluate *Hive.js* [19], a WebRTC-based, peer-assisted video streaming solution. A central tracking server keeps track which player instances are currently connected. Their software then builds a random graph between all viewers of a video. Periodically, each peer selects one of its WebRTC connections and terminates it. Immediately after that, the peer connects to a new randomly chosen peer from a list of potential new neighbors. This list is provided by the tracking server. An evaluation has shown that their system performs well for peer-to-peer streaming with 30 or less peers. Hive.js does not check data integrity of downloaded video fragments.

Nogueira Barbosa et al. describe an implementation of a WebRTC-based, peer-assisted video streaming solution [12]. The system uses an ISP location and geolocation awareness concept to build clusters of nodes. Each peer in their system belongs to a WebRTC cluster. Peers who belong to the same provider's network and who are geographically close are preferably assigned to the same cluster. All peers in the same cluster are directly connected to each other through WebRTC connections. If a peer needs a media fragment, it first tries to receive it via broadcasting the request within its cluster. If the peer is not able to receive a desired media fragment from the peer-to-peer network in time, it requests it from a CDN. The authors' experiments demonstrate that their implementation leads to high fluctuations regarding the percentage of fragments that are delivered through the peer-to-peer network.

Meyn describes another WebRTC-based, peer-assisted video streaming system [8]. Due to the fact that at the time of conceptualization no browser vendor had implemented support for the WebRTC data channel, no prototype was created. Nurminen et al. describe a WebRTC-based, peer-to-peer video streaming system [13]. Similarly, no working WebRTC data channel was available on mainstream browsers. Instead, the authors evaluated the performance of the MD5 hashing algorithm that is used to validate data integrity of media fragments that were delivered by peers.

Finally, in addition to the related work mentioned above that deals with peer-to-peer video streaming systems similar to our goals, we would like to highlight work that examines general quality aspects of WebRTC-based video streaming in the browser. In particular, the MONROE[2] platform, a European testbed for such experiments, can be mentioned. Moulay et al. use it to analyze the applicability of WebRTC in mobile scenarios [11]. The results are consistently positive, but the authors mention quality leaps in mobile networks due to insufficient network coverage. On the same platform Sulema et al. analyzed the *Quality of Experience* [24]. They come to the conclusion that the video transmission quality of WebRTC is not inferior to that of platform-specific commercial solutions.

2.4 Delivery Models

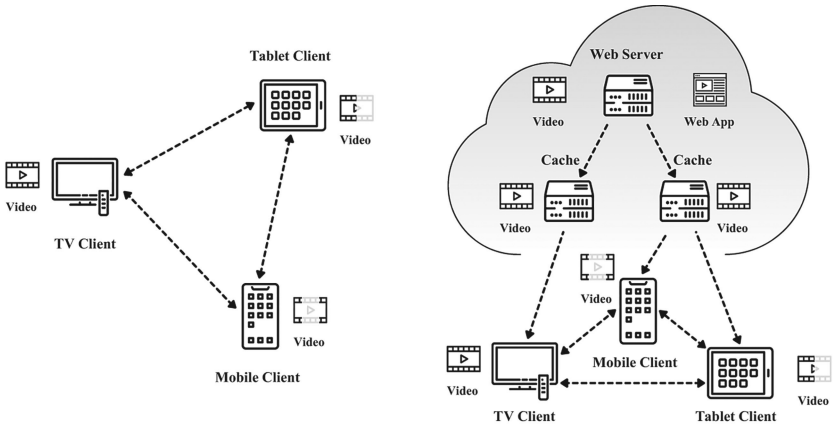
Video content delivery as it is currently implemented on various popular websites like YouTube, Facebook and Vimeo follows a client-server model. Users upload the video to a central cloud repository. When other users retrieve the video's website, the provider embeds the URL of the video in the returned HTML page, typically within a `<video>` tag. The browser then requests the video from the server over the specified URL. After that, the server initializes the video stream to the client. In this scenario, the video is always delivered from the server to the client. No data is transferred between any two clients. Although Web caches such as CDNs can reduce the load on the main server, even if a video is watched at the same time on two different client devices side by side, two separate connections to the server are established and the data is downloaded twice. The limitation of the number of requests a server can answer is a significant bottleneck of the client-server architecture. However, traditional client-server systems can easily ensure content integrity and reliable accounting.

By contrast, the advantage of peer-to-peer systems compared to client-server models is that the load is decentralized onto the clients. Peers are connected to each other and ideally retrieve video fragments from neighbor peers. The disadvantages of peer-to-peer systems

[2]https://www.monroe-project.eu/

compared to sufficiently equipped client-server systems are longer initial start-up times as the video source has to be negotiated initially, and more unwanted stuttering or stalling of video play due to the disappearance of other peers during the playback [6]. In a peer-to-peer network, the time span between connecting and receiving the first byte of a media fragment is relatively large; getting the first fragment from a Web server or CDN is significantly faster, as the client has a fixed endpoint with which it can negotiate the byte stream.

Peer-assisted video streaming is another delivery method. It is a hybrid model that combines the advantages of client-server like high availability with the load distribution of pure peer-to-peer system [6, 12, 15]. In a peer-assisted streaming system, if a peer cannot receive a desired media fragment from the peer-to-peer network in time, it downloads it from the source of the media stream, e.g. from a server, like Hive.js does [19]. The short initial start-up times are possible because in this kind of hybrid solution the first fragments of media content can be downloaded from the CDN or Web server to enable the start of the video playback as soon as possible. This fallback solution guarantees that all media fragments are received by each peer early enough to avoid stuttering or stalling of video playback. Figure 2 compares both peer-to-peer and peer-assisted streaming architectures.



(a) Peer-to-Peer (b) Peer-Assisted

Figure 2 Comparison of peer-to-peer video streaming architectures.

3 Requirements for Peer-to-Peer Video Delivery

Considering the available body of research literature described in Section 2, we present a number of functional and non-functional requirements for our system in the following. As of today, all browsers including the initially hesitant Apple Safari support the WebRTC or similar RTC group of standards for peer-to-peer connections on the Web[3]. The Edge browser by Microsoft currently exposes the *Object RTC API* with similar capabilities [9]; thus the findings of our system should be easily transferable.

Generally, we want to enable streaming videos in a peer-to-peer manner between instances of different browsers. Random access of playback positions should be allowed to enable users to jump freely during the stream. There are different strategies for distributing the video content amongst the peers. *Rarest piece selection* means that those fragments of a video get requested first, that are estimated as rarest in the network. Offloading streams from the cloud to a peer-to-peer network naturally shifts the load onto the clients. We therefore envision having a fair distribution of the load amongst the peers. A ratio defining the proportion of download and upload should be configurable in addition. For example, it is more reasonable to stream from a device that is connected to a stable wired connection than it is from a battery-powered mobile device in a slow mobile network. The respective configuration parameter is the *peer upload limit*. Other parameters include the buffer size, which is the length of seconds of video playback we cache to enable a seamless playback of the video file. Finally, the system to be developed should generate and expose statistical values for further tuning the system. For example, the amount of video data downloaded from other peers and the amount of uploaded data could be logged. The connection to other peers should either be mediated by a central server or be established via a manual connection.

Finally, non-functional requirements include the ability to cope with high fluctuations of the available bandwidth. Also, the system should be quite resistant to continuous arrival and departure of peers, and accommodate some redundancy. This means that the video stream

[3]https://caniuse.com/#feat=rtcpeerconnection

must not slow down or stop completely if a peer leaves the network. However, we assume a stable connection of the seeding peer. The initial availability of the video thereby depends on the seeder's bandwidth. We particularly stress the importance of a comprehensible and easy-to-understand documentation of the source code, as the results should be available as open source solution to foster further development through the wider open source community.

Table 1 compares the mentioned related work and discusses properties that the systems fulfill or not. We additionally added the following commercial solutions to the matrix: StreamRoot[4], Viblast[5], Peer5[6] and Swarmify[7]. To the best of our knowledge, there is no major conceptual difference between these four solutions. All of them use a tracking server, advertising that their solution is effective in choosing the best

Table 1 Comparison of WebRTC-Based Video Streaming Frameworks

	Framework								
System Property	Hive.js [19]	Nogueira Barbosa et al. [12]	Meyn et al. [8]	Rhinow [18]	StreamRoot	Viblast	Peer5	Swarmify	**OakStreaming**
Working implementation	●	●	○	●	●	●	●	●	●
Detection of poisoned fragments	○	○	●	○	●	●	●	●	●
Small scale efficiency	●	○	–	–	●	●	●	●	●
Intelligent peer selection	○	●	○	○	●	●	●	●	●
Upload limit per peer	○	○	○	○	●	–	–	–	●
Open source (code/ algorithm)	●	●	●	○	○	○	○	○	●

● = provides property; ○ = does not provide property; – = unknown

[4]https://streamroot.io/
[5]https://www.viblast.com/
[6]https://www.peer5.com/
[7]https://swarmify.com/

peers for data sharing based on geography and network topology. The four systems use WebRTC signaling servers due to the characteristics of the protocol. The common factor between these companies is that the algorithms are not available open source.

Out of the academic solutions, all but one have a working implementation. None are able to effectively capping the outgoing connections after a video has been entirely downloaded by limiting the upload per peer. In our OakStreaming library, we aim to fulfill all these properties.

4 Conceptual Design

To create a Web-based peer-to-peer solution for distributing video data, we evaluated three different architectures: a synchronized look-up table, a distributed hash table and a tracking server. In this section, we share our reflections and discuss their implications.

4.1 Synchronized Look-Up Table

The goal of a synchronized look-up table concept is to collect all available information about the current playback and buffer state of each peer in the network. In this approach, peers publish which fragments they have cached already. Peers use this information to update their own look-up tables where they keep track of the current overall video distribution state. A simple algorithm could now ask all peers who have already loaded a required fragment to pass the fragment. Another possibility is to send out a message to all other peers together with an urgency indicator calculated out of the temporal distance of the currently played fragment. Other peers then react to the message and offer the needed fragment. As a consequence, there is a large number of messages that need to be broadcasted to all participants, thus any naive algorithm would not be scalable to a large number of peers. A solution could be to multicast messages to a subset of peers. This practice of limited information exchange proved efficient in the WebRTC-based peer-to-peer video streaming system Hive.js [19]. It uses a so-called tracking server to create a random graph between peers. Each peer only

queries peers for fragments to which a direct peer-to-peer connection already exists.

4.2 Distributed Hash Table

The main aim of a distributed hash table (DHT) concept is to make finding a peer which can deliver the desired media fragment possible, reliable and efficient without a need for a central coordination node. Distributed hash tables are decentralized distributed systems which provide a look-up service for key-value pairs [17]. DHTs have a network structure consisting of nodes and connections between these nodes. The look-up functionality of a DHT gets as input an arbitrary key out of a defined key space, and outputs the corresponding value. Each key is (temporarily) assigned to a network node and any participant can retrieve the corresponding value by sending a query message into the DHT network which then gets routed to a node that got assigned this key. Responsibility for the key space is distributed among all nodes of the network in such a way that the quality of service provided by the DHT is robust against continuous node arrivals, departures and failures. Moreover, DHTs do not have a single-point of failure, which enables peer-to-peer video streaming systems that use a DHT to be fairly safe against total failures. A popular DHT algorithm and protocol is Chord [22]. Every node of a Chord network has to maintain not more than $O(\log n)$ connections to other nodes of the network, whereby n is the number of nodes of the DHT. Chord's connection layout and routing algorithm make it possible to route to an arbitrary node of the DHT in $O(\log n)$ hops, whereby n again is the number of nodes of the DHT. Using the key-value storage functionality of DHTs, it can be dynamically stored and retrieved which peer can deliver which video fragments. The mentioned properties of Chord make it possible to find a peer that provides a desired media fragment in an efficient and scalable way. Most DHT concepts have similar properties like Chord, making them attractive as a basis for peer-to-peer video streaming systems.

4.3 Tracking Server

Another possible solution to implement a peer-to-peer video streaming system is to connect every peer to a central tracking server that stays in contact with each peer to keep an overview of the network state. No actual video data is transferred from or to a tracking server. For performance reasons, the tracking server could observe, (a) which peer needs which media fragment in the next few seconds, and (b) how reliable each peer has been in the last seconds. On the one hand, a tracking server may calculate which peer should best send which media fragments to which other peer and then send the corresponding orders to the peers. On the other hand, it can also easily organize which peer connects to which peer and let the peers send requests for media fragments to other peers. The latter approach highly reduces organizational efforts of the tracking server and therefore significantly improves the scalability. In [12] as well as in [19], this tracking server approach has proven successful. Peer-to-peer video streaming concepts based on tracking servers are tried and tested, which is shown by the fact that every WebRTC-based, peer-to-peer video streaming implementation presented in Section 2.3 uses a tracking server. An obvious disadvantage of the tracking server concept compared to the two aforementioned main concepts is that the tracking server is a single point of failure. When the tracking server stops working, no peer can find new peers to connect to.

A *torrent tracker* is a special kind of tracking server. The torrent concept works with torrent files each of which groups other files into fragments and identify them by cryptographic hash values. Furthermore, it lists the size (in bytes) of the fragments. Moreover, a torrent file optionally contains additional information such as one or more URLs to torrent trackers. Anyone who knows the hash value of the entire torrent file can request direct peer-to-peer connections from tracking servers to peers that want to exchange fragments of that file. Torrent trackers, additionally, enable to build up peer-to-peer connections between two peers that are interested in the same torrent file, taking the role of a signaling server. The protocol defines how peers request file fragments from each other. The set of peers with whom a peer shares a direct peer-to-peer connection is called the *swarm instance* of the peer. Peers

only request file fragments from peers with whom they have already established a peer-to-peer connection. A torrent file can be tracked by multiple tracking servers. With the hash value contained in the torrent file, any peer can check the integrity of received fragments.

4.4 Discussion and Concept

The preceding discussion of possible concepts for the planned peer-to-peer system revealed that the concept of a synchronized look-up table scales significantly worse than DHT or a tracking server concept. Related work shows that tracking server concepts are quite well studied and work reasonably. DHTs, on the other hand, do not have a single point of failure and new peers only need one connection to any node of the DHT to enter the DHT network. Therefore, using a DHT seems to be a viable solution. Unfortunately, to the best of our knowledge there is no solid DHT implementation that runs in the browser without a plugin; developing a new one was out of the scope of this work.

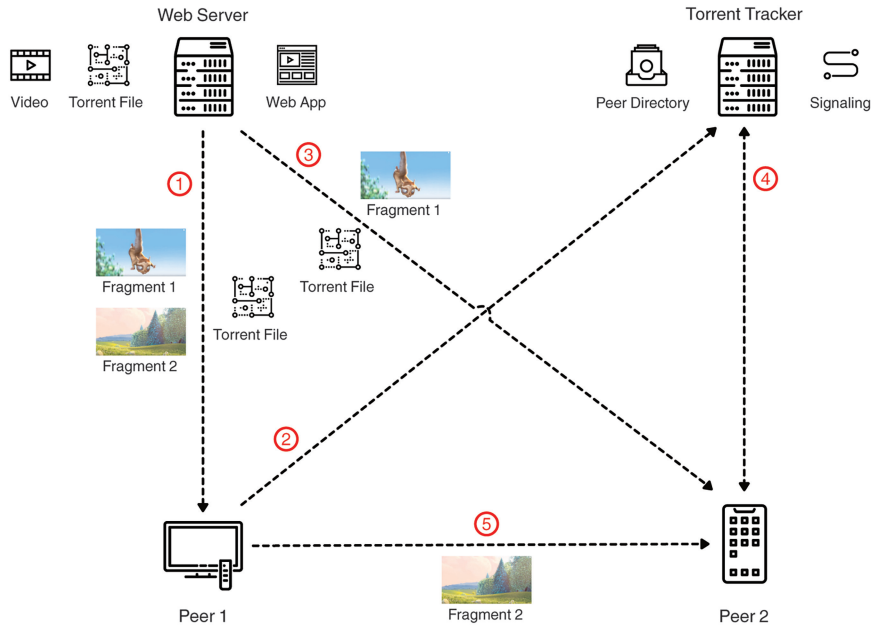


Figure 3 Overview of proposed architecture and video fragment exchange.

In the following, we therefore present our solution based on a torrent tracker, named *OakStreaming*. Figure 3 shows the architecture of our system including its three main parts: the Web server delivering the initial Web application including the OakStreaming client-side library, the WebTorrent tracker responsible for sharing video fragment locations, and the peers interested in watching the video. First, peer 1 retrieves the Web application and the initial video fragments from the Web server (1). The server also keeps a torrent file which includes information about the fragments of the video. The client then announces the availability of fragments to the torrent tracker (2). If a second peer connects to the system by retrieving the Web application (3), subsequent fragments are already available on the first peer. Therefore, the torrent tracker announces the availability of fragment 2 to the second peer (4), which then starts a peer-to-peer connection to the first peer (5). The torrent tracker is also responsible for negotiating the direct peer-to-peer data channel between the peers (in the signaling phase of WebRTC).

The example above showed the simplest case for retrieving video from the peer-to-peer network. We additionally include several strategies for efficient data transfer. For instance, various parameters can be set to limit the amount of uploaded data from a peer; in the example above, the second peer could have already retrieved the first fragment from the peer-to-peer network. Alternatively, the video file can also be obtained from a participating peer. We provide two strategies for downloading video fragments. The first, *sequential-piece-selection*, downloads fragments sequentially based on the timeline of the video. The second strategy, *rarest-piece-selection*, first retrieves those fragments, that are most needed throughout the entire peer-to-peer network. The concrete library methods and their parameters are explained in the next section.

5 OakStreaming Peer-to-Peer Video Streaming Library

In this section, the implementation of the video streaming framework is presented. The system consists of a WebTorrent tracker, a Web server and OakStreaming instances. The OakStreaming instances run on the devices of the viewers.

The implementation is based on the WebTorrent JavaScript library. It is an adaptation of the BitTorrent protocol for the Web, using WebRTC connections for exchanging data fragments. We extended its functionality significantly by introducing additional parameters targeting video streams. The OakStreaming library has been developed as a Node.js module which is turned into a Web browser compatible version via the Browserify[8] bundler. The Node.js module only exports a single object which is the constructor to create OakStreaming instances; it is global to the browser window's namespace. An OakStreaming instance provides several public methods for the library user.

5.1 Initiating a Stream

The `createStream(callback, videoFile, options)` method expects one required and two optional parameters. The required parameter is a callback function which gets called with a previously instantiated `StreamTicket` object that contains all the streaming properties. The first optional parameter is the video file which is handed over as a W3C File object. The second optional parameter contains options for the streaming process. If a video file gets handed over to the `createStream` method, it seeds this video file to the WebTorrent network. In this case, additional streaming information is entered through the `options` argument.

5.2 Receiving a Stream

Most logic of our OakStreaming library is hidden behind the `receive Stream (streamTicket, callback, capSeeding)` method. The required `streamTicket` argument expects a `StreamTicket` object containing peer and Web server connection information. An OakStreaming client can download a video from a Web server and from peers of the WebTorrent network in parallel. The `callback` function gets called upon successful connection. The `capSeeding` boolean parameter defines whether upload to other peers should be terminated once the video is fully downloaded locally. To be able to play back

[8] http://browserify.org/

all common video formats, each peer repacks received media fragments on-the-fly by using a JavaScript module called videostream[9]. A videostream object repacks media fragments and puts them into a source buffer to be played back by a HTML5 `<video>` element.

As soon as the WebTorrent instance has processed a torrent file and found peers, it starts downloading the video in the background according to the rarest-piece-selection strategy. The torrent tracker also exchanges the signaling data amongst the peers.

Byte range requests can be conducted by calling the `createReadStream()` method of the WebTorrent API. This method returns a readable stream object which can be used to (partially) read the requested byte range even if it has not been downloaded completely yet. File data is made available by the readable stream as soon as it is received. If the byte range request should not span the entire file, the range can be specified through an argument; the start and end properties are inclusive. Byte range requests which were created from calls to `createReadStream` are fetched as fast as possible and in sequential order from the WebTorrent network. The rarest-piece-selection fragment downloading, as initialized by the creation of a WebTorrent instance, is suspended as long as a stream returned by `createReadStream` has not yet received every byte out of its byte range request.

So far we have discussed the ability to download the video to be streamed as referenced in an existing torrent file hosted on a Web server. However, we also provide the possibility to initiate the streaming of a video file from a participating peer by creating a new torrent file. Figure 4 shows the sequence of messages sent around between OakStreaming peers and the torrent tracker, in the case that a peer intends to stream a local video file to other clients. First, a torrent file is created on the first peer by calling the library with a local video file. The library then registers the peer as seeder of the video file. To notify the second peer about the availability of the file, a `StreamTicket` including the torrent file is sent to the second peer. The second peer then queries the tracker together with its signaling data. It is forwarded to the first

[9] https://github.com/jhiesey/videostream

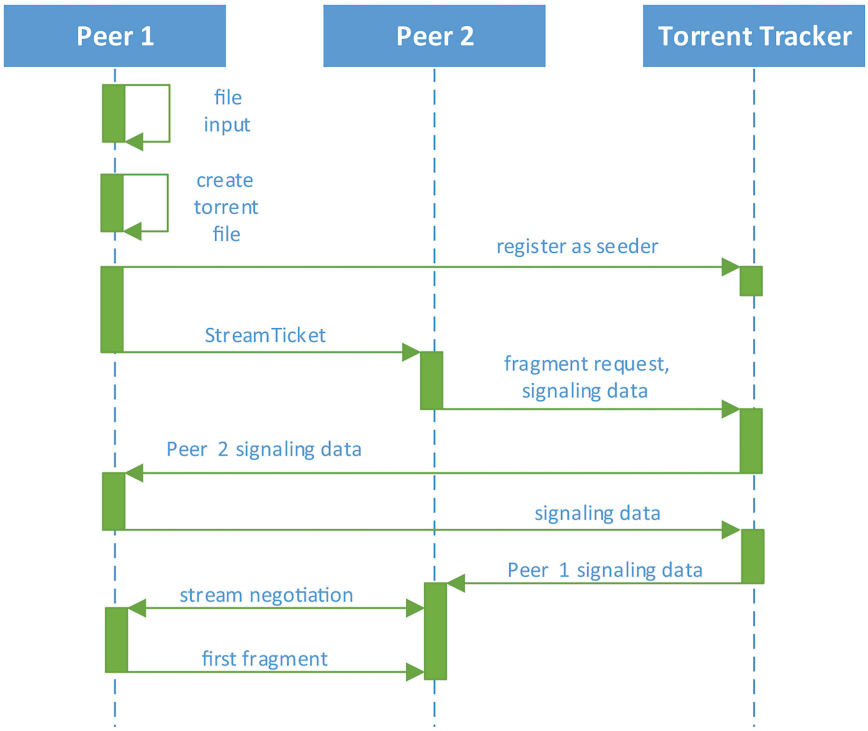


Figure 4 Establishing an OakStreaming peer-to-peer session.

peer, asking for its signaling data in turn. After the first peer's signaling data is presented to peer 2, both start a WebRTC stream negotiation process. Finally, the first fragment of the video is transfered from peer 1 to peer 2.

Listing 1 shows the exemplary usage of the OakStreaming library within an HTML5 Web page, implementing the sequence described above. Note, that this is pseudo-code, and no sanity checks or exception handlers are included. The library is loaded as a JavaScript module on line 4. Line 7 shows a file `<input>` field, with the event handler being called as soon as the user selects a file in the system dialog. In line 8, a standard HTML5 `<video>` tag is shown. In the `<script>` tag, first the library is loaded. The event handler on line 13 is called once a video file is loaded on peer 1. Therein, a configuration object is created that includes a Web server for peer-assisted delivery and

```
<!doctype html>
<html>
<head>
  <script type="module" src="oakstreaming.js"></script>
</head>
<body>
  <input type="file" onchange="handleFiles(this.files)">
  <video id="video"></video>
  <script>
    let oakStreaming = new OakStreaming();

    // code executed at Peer 1
    function handleFiles(files) {
      let config = {web_server_URL: 'localhost:8080',
        webTorrent_trackers: ['ws://localhost:8085']};
      oakStreaming.create_stream(files[0],
        config,
        streamInfo => {
          // share stream information with other peers
          notifyPeers(streamInfo);
        });
    }

    // code executed at Peer 2
    function handleNotification(streamInfo) {
      oakStreaming.receive_stream(streamInfo,
        document.getElementById('video'),
        _ => {
          // all video data has been received
        });
    }
  </script>
</body>
</html>
```

Listing 1 HTML5 pseudo-code of an OakStreaming library client.

an initial WebTorrent tracker URL. Then, the stream is created with the respective OakStreaming method. The first argument is the `File` object returned from the `<input>` field, the second is the configuration, the third is a callback function. The latter is called, once the torrent

file has been created. In this example, we call the fictional method `notifyPeers(streamInfo)` that sends the stream information to remote peers. It informs peer 2 via the function on line 25. It receives the same stream info property previously supplied in the callback method. To start the stream to the local `<video>` tag, again an OakStreaming method is involved.

The library allows to log various parameters of the stream. These include the overall number of bytes downloaded or uploaded from other peers, as well as the ratio of bytes downloaded from a peer-assisting server vs. the bytes retrieved from other peers.

6 Evaluation

The evaluation of our library is divided into a technical analysis and a user study. In the technical part, we measured how the three implemented features added on top of the WebTorrent library affected the video streaming. In the user evaluation, we asked seven Web developers to use the OakStreaming API and give feedback regarding the usability of the library, its documentation and peer-to-peer video streaming in general.

6.1 Technical Evaluation

In order to test how well the implemented system is able to organize peer-to-peer streaming, several tests have been conducted with up to eight peers. The tests were run on a middle-end laptop running the Windows 10 operating system and the Chrome browser using the Blink browser engine. Each test was conducted with one seeding OakStreaming instance and 2, 4 or 8 OakStreaming instances which emulated viewers of the video. The video used was a three minute high definition (HD) video that comprised 106 Megabytes. At the start of each test, each OakStreaming instance established a Web-Socket connection to a WebTorrent tracker. As implemented in the WebTorrent library, the tracker automatically initializes the WebRTC connection between OakStreaming instances. With two WebTorrent instances connected, they can exchange video fragments. We have

found that there are a number of factors that cause a delay of up to a few seconds in the initialization phase of the stream over WebTorrent. The process of querying a neighbor for video fragments could take a significant amount of time, even if the peer-to-peer connection had already been established. Moreover, a peer can only receive file data from its neighbors of fragments whose size is specified during creation of the torrent. These circumstances increase the time from the moment a peer receives a chunk of video data to the moment it serves the received chunk to the viewer. The OakStreaming library uses the default hash value calculation algorithm of the WebTorrent library for the fragment. Additionally, after a WebRTC connection between two peers has been established, the WebTorrent protocol needs time to initialize the neighborhood conditions and exchange information which fragments which peer can deliver. Because of the delays, the emulated viewers were started with a random time offset. First, an interval from 0 to 10 seconds was chosen. Using this interval, the average amount of video data that the viewers delivered to each other was relatively low. Therefore, several interval sizes were tested. Besides 0 to 10, the checked interval sizes were 0 to 20, 0 to 30 and 0 to 60. When using 0 to 60 as time offset interval, most data was transferred between emulated viewers compared to the other three; it was therefore chosen as the offset value for all test runs.

As a result, our tests showed that a lower threshold for the video playback buffer before the OakStreaming client switches from sequential-piece-selection to rarest-piece-selection leads to a higher overall download time. This correlation was expected and confirms the usefulness of the parameter, as with a longer sequential setting, rare fragments are prioritized lower, possibly leading to bottlenecks with video progression. Depending on the scenario, different values may be useful.

We also tested pure peer-to-peer delivery versus peer-assisted delivery regarding the average time the playback was stalled. Figure 5 shows a comparison of pure peer-to-peer and peer-assisted streaming in regard to the total amount the video was stalled, as an average of multiple test runs. As expected, peer-assisted delivery significantly reduces stalling

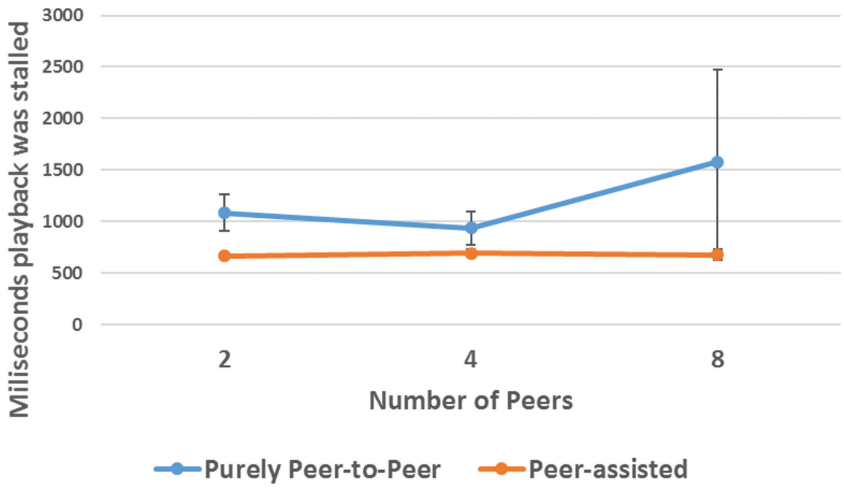


Figure 5 Peer-to-peer vs. peer-assisted streaming.

time; here, video playback was only interrupted during initial start-up (around 600 ms).

The results of the technical evaluation clearly indicate that peer-assisted delivery and automatic switching between sequential-piece-selection and rarest-piece-selection enhance the overall quality of service of the peer-to-peer video streaming system, when taking the whole network into consideration. When the peer-to-peer network consisted out of two peers, the average start-up time for pure peer-to-peer streaming was 1086 milliseconds. In case of peer-assisted streaming this number went down to 667 milliseconds. Measurements with four peers in a pure peer-to-peer environment resulted in an average start-up time of 936 milliseconds. In case of peer-assisted streaming this number went down to 695 milliseconds. Overall, we are convinced that for the discussed use cases, the added delay of around 400 ms is acceptable. In conclusion, we can say that the applicability of the library depends very much on the usage scenario. For a comprehensive discussion of the implications for various application scenarios, see Section 7.1.

6.2 Developer Evaluation

The aim of the developer evaluation was to test the usability and acceptance of the OakStreaming library by asking potential library users for their opinion about the design of the library and peer-to-peer streaming in general. The seven participants drawn out of the pool of student employees at our department had intermediate to advanced skills and experience in the areas of torrent-based peer-to-peer systems, peer-to-peer systems in general, video streaming/hosting and JavaScript. The lab experiment comprised three programming task and filling out the evaluation questionnaire. The programming tasks were designed to make the participants familiar with the API and functionality of the OakStreaming library. The questionnaire mainly aimed at collecting data about the usability and documentation of the OakStreaming library as well as general opinions regarding peer-to-peer video streaming.

6.2.1 Session setup and programming tasks

The participants were invited in groups of two to three and had to solve the same three tasks, but they were asked to work on them individually. The setup was the same for all three programming tasks. The final Web application of each task should be tested in two to three Web browser windows. The participants could conduct these tests independent from each other on their own device.

The first programming task was to create a Web application which uploads and downloads a video to and from a Web server. The second and third programming tasks then both focused on completing the program code of a Web application which streams a video peer-to-peer. The peer-to-peer connections were established locally between the browser windows on the devices of the participants.

In task 2, the participants had to use the `streamVideo` method of the OakStreaming library to create a `StreamTicket` object from a video file. The object was then shared over the a synchronized data structure with the Yjs collaboration library[10]. The received object was then put into the `receiveStream` method of the peer instances.

[10] http://y-js.org

Task 3 was very similar to task 2 but the participants had to use different parameters and parameter values when creating the `StreamTicket` object. In contrast to task 2, the video should be streamed in the peer-assisted mode and the participants had to set a value for the ratio of time downloaded from server versus peer-to-peer. The parameter values of task 2 and 3 were given by the task description. To solve task 3 it was necessary to read parts of the OakStreaming documentation. Finally, an evaluation form was filled out by the participants. It asked the participants about their knowledge and experience in the areas of torrent-based peer-to-peer systems, peer-to-peer systems in general, video streaming/hosting and JavaScript. Moreover, the participants were asked to rate the usefulness of several features that the OakStreaming library implemented on-top of WebTorrent. Additionally, the participants had to rate the usability of the OakStreaming API documentation. Furthermore, the participants were asked about their opinion regarding peer-to-peer video streaming in general.

6.2.2 Results

Five of the seven participants were able to solve all three tasks. Two participants were only able to complete the tasks after short assistance. The questionnaire revealed minor issues with the arguments, like putting together URL and port properties. After the evaluation the respective API was changed to a single URL property instead, which can now handle strings in several formats (e.g. http://example.com:42, http://example.com, example.com:42, etc.). Overall, all features of the OakStreaming library were considered easily understandable.

We were also interested in general opinions of our developers on peer-to-peer video streaming. Most of the participants rarely publish or share, but often consume Web videos. They saw many important advantages of peer-to-peer video streaming like benefits for small content providers with successful videos and the breaking of the monopoly of large content providers in terms of intellectual property rights. Additional remarks of the respondents covered legal issues if possibly illegal videos are streamed between peers.

7 Discussion and Future Work

This paper presented OakStreaming, a peer-to-peer video streaming system for the Web. It is available as open source software on GitHub[11]. The three main motivations for developing it were to reduce server load compared to state-of-the-art client-server-based video streaming systems; to avoid transfer of intellectual property to a third party; and to maintain a reasonable quality level for the viewer. Since all major browser manufacturers have implemented the W3C WebRTC or similar specifications, some commercial WebRTC-based peer-to-peer video streaming systems have been developed, while solutions by the academic community lack many desired features. We discussed the three alternative concepts synchronized look-up table, distributed hash table and torrent tracker. Finally we implemented the WebTorrent based peer-to-peer video streaming system called OakStreaming. Our evaluation shows that the peer-assisted modus representing a hybrid scenario with peer-to-peer video delivery and a fallback server significantly reduces video playback start-up time. Peer-to-peer video streaming functionality has been implemented based on the content delivery functionality of the WebTorrent library.

The OakStreaming library extends the WebTorrent library by various means:

- configurable limit on the amount of data each peer is allowed to upload
- configurable parameter specifying when the client switches from sequential-piece-selection to rarest-piece-selection
- dynamic combination of server and peer-to-peer streaming (peer-assisted delivery)
- possibility to easily add new client instances to an existing peer-to-peer network, without using a torrent tracker, by explicitly exchanging signaling data

The main challenge during the implementation was that in many third-party libraries we employed, documentation was lacking. Some properties were only explained implicitly in GitHub issues and not in the official API description. This may be due to the fact, that the

[11] https://github.com/rwth-acis/WebTorrent-VideoStreaming/

corresponding standard is still very new, and the libraries are not yet fully mature.

The results of the technical evaluation clearly indicate that peer-assisted delivery and automatic switching between sequential-piece-selection and rarest-piece-selection enhance the overall Quality of Service (QoS) of the peer-to-peer video streaming system. In the developer evaluation, the usability and the design of the OakStreaming library were positively assessed by the participants. Moreover, the results of the questionnaire have shown that all participants agreed that peer-to-peer video streaming will become more important in the future. The results are helpful to set further goals regarding research in the area of Web-based peer-to-peer video streaming. Limitations of our work include further evaluation in larger developer and user groups. With regard to fault tolerance, our system behaves differently depending on which part has gone down. If the original seeder fails, while not all fragments have been transferred yet, these video parts cannot be compensated accordingly. Possibly a conservative approach with a caching server can be more appropriate to deal with these cases. However, as long as the WebTorrent tracker is online, the remaining fragments will continue to be distributed. Privacy aspects in terms of sharing peer information were neglected by our research, but we are closely following security issues raised by the introduction of WebRTC in the browser [14, 16].

7.1 Use Case Recommendations

We conclude the discussion of our library with some recommendations for its employability in different use cases. Generally, our library is best for videos that are highly demanded within a certain period of time. Because the code is no longer executed after leaving a Web page, the client can no longer upload fragments. While it might be possible to work around this by deploying JavaScript Service Workers, other conflicting effects, such as the increasing storage space requirements of large video files or limited mobile data plans, play a detrimental role. Table 2 shows typical use cases of video streaming on the Web. We mention examples as well as an applicability estimation of our library.

Table 2 Use Case Recommendations for the OakStreaming Library

Use Case	Examples	Applicability
Live broadcast/ multicast	Sports, Live TV	Moderate applicability due to WebTorrent concept
Time-limited asynchronous broadcast (up to 1 day)	Short videos with temporal high demand, social media sharing, TikTok, Instagram Stories	Fully applicable
On-demand with temporal bursts	Educational videos, MOOCs	Applicable, depending on mobile bandwidth
On-demand	YouTube, Netflix, Amazon Prime	Limited applicability, best for highly-demanded videos (e.g. for releases of new episodes).
Video conferencing	Skype, Webex	Not applicable due to WebTorrent concept

7.2 Future Work

We are planning to embed the library in various Web application prototypes to measure long-term effects and performance. In doing so, we plan to target the usage scenarios mentioned in Table 2. In general, we believe that an open collection of best practices is needed on the applicability of Web-based peer-to-peer technologies depending on the use case. Open source solutions that make the various approaches publicly testable are also in demand. Generally, we deem WebTorrent applicable for further application areas besides video streaming where large chunks of data need to be transfered to multiple clients. For example, it can be used for dealing with 3D content in augmented or virtual reality scenarios (cf. [5]). Further applicable targets are level 5 peer-to-peer Web applications in the context of *liquid* Web applications [4].

Today's Web videos are often streamed with an adaptive bitrate streaming technology, which downloads sections of the same video in different bitrate versions depending on the network conditions of

the client. The DASH (Dynamic Adaptive Streaming over HTTP) technique is an international standard that enables adaptive bitrate streaming with conventional HTTP servers. Implementing adaptive bitrate streaming within a peer-to-peer network with a satisfying viewer experience is not trivial, as respective fragments would have to be available or encoded on-the-fly on peers. To the best of our knowledge, there is no implementation which offers an adaptive bitrate in a WebRTC-based peer-to-peer video streaming system. Therefore, adding support for adaptive bitrate streaming to the OakStreaming library is a promising aim for future research and development.

Acknowledgements

We thank our student Philipp Bartels for his contributions towards the implementation of the prototype and we are grateful for the feedback received in our evaluation. We would also like to thank our anonymous reviewers; the valuable comments helped to shape the article. The work has received funding from the European Commission's FP7 IP "Learning Layers" under grant agreement no. 318209 and from the European Research Council under the European Union's Horizon 2020 Programme through the project "WEKIT" (grant no. 687669).

References

[1] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, Bernard Aboba, Taylor Brandstetter, and Jan-Ivar Bruaroey. WebRTC 1.0: Real-time Communication Between Browsers: W3C Candidate Recommendation 27 September 2018.

[2] Rajkumar Buyya, Mukaddim Pathan, and Athena Vakali. *Content Delivery Networks*, volume 9 of *Lecture notes in electrical engineering*. Springer-Verlag, Berlin, 2008.

[3] Cisco Systems. The Zettabyte Era: Trends and Analysis: June 2017.

[4] Andrea Gallidabino and Cesare Pautasso. Maturity Model for Liquid Web Architectures. In Jordi Cabot, Roberto de Virgilio, and Riccardo Torlone, editors, *Web Engineering*, volume 10360, pages 206–224, Cham, 2017. Springer International Publishing.

[5] Yonghao Hu, Zhaohui Chen, Xiaojun Liu, Fei Huang, and Jinyuan Jia. WebTorrent Based Fine-grained P2P Transmission of Large-scale WebVR Indoor Scenes. In Matt Adcock and Tomasz Bednarz, editors, *Proceedings of the 22nd International Conference on 3D Web Technology – Web3D '17*, pages 1–8, New York, USA, 2017. ACM Press.

[6] B. Li, S. Xie, Y. Qu, G. Y. Keung, C. Lin, J. Liu, and X. Zhang. Inside the New Coolstreaming: Principles, Measurements and Performance Implications. In *IEEE Conference on Computer Communications*, pages 1031–1039. IEEE, 2008.

[7] Bo Li and Hao Yin. Peer-to-Peer Live Video Streaming on the Internet: Issues, Existing Approaches, and Challenges. *Communications Magazine, IEEE*, 45(6):94–99, 2007.

[8] A. J. Meyn. *Browser to browser media streaming with HTML5: Master's thesis*. Aalto University, 2012.

[9] Microsoft. Object RTC API, 2017.

[10] Christian Mossner and Dennis Herhausen. Video: the New Rules of Communication. *Marketing Review St. Gallen*, 34(2):36–44, 2017.

[11] Mohamed Moulay and Vincenzo Mancuso. Experimental Performance Evaluation of WebRTC Video Services over Mobile Networks. In *2018 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 541–546. IEEE, 15.04.2018–19.04.2018.

[12] Flávio Ribeiro Nogueira Barbosa and Luiz Fernando Gomes Soares. Towards the Application of WebRTC Peer-to-Peer to Scale Live Video Streaming over the Internet. In *Simposio Brasileiro de Redes de Computadores (SBRC)*, 2014.

[13] Jukka K. Nurminen, Antony J. R. Meyn, Eetu Jalonen, Yrjo Raivio, and Raúl Garcıa Marrero. P2P Media Streaming with HTML5 and WebRTC. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 63–64, 2013.

[14] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis.

Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. *CoRR*, abs/1810.00464, 2018.

[15] Konstantina Papagiannaki, Krishna Gummadi, Craig Partridge, Mingchen Zhao, Paarijaat Aditya, Ang Chen, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, Bill Wishon, and Miroslav Ponec. Peer-assisted content distribution in Akamai netsession. In *IMC '13 Proceedings of the 2013 conference on Internet measurement conference*, pages 31–42. 2013.

[16] Andreas Reiter and Alexander Marsalek. WebRTC: Your Privacy is at Risk. In *Proceedings of the Symposium on Applied Computing*, pages 664–669. ACM, Marrakech, Morocco, 2017.

[17] Eric Rescorla. Introduction to Distributed Hash Tables, 2006.

[18] Florian Rhinow, Pablo Porto Veloso, Carlos Puyelo, Stephen Barrett, and Eamonn O. Nuallain. P2P live video streaming in WebRTC. In *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, pages 1–6, 2014.

[19] Roberto Roverso and Mikael Hogqvist. Hive.js: Browser-Based Distributed Caching for Adaptive Video Streaming. In *IEEE International Symposium on Multimedia*, pages 143–146, 2014.

[20] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge Analytics in the Internet of Things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[21] Thomas Stockhammer. Dynamic adaptive streaming over HTTP – Standards and Design Principles. In Ali C. Begen and Ketan Mayer-Patel, editors, *Proceedings of the second annual ACM conference on Multimedia systems – MMSys '11*, page 133, New York, USA, 2011. ACM Press.

[22] 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 Transactions on Networking*, 11(1):17–32, 2003.

[23] Masayuki Sugawara, Seo-Young Choi, and David Wood. Ultra-High-Definition Television (Rec. ITU-R BT.2020): A Generational Leap in the Evolution of Television [Standards in a Nutshell]. *IEEE Signal Processing Magazine*, 31(3):170–174, 2014.

[24] Yevgeniya Sulema, Noam Amram, Oleksii Aleshchenko, and Olena Sivak. Quality of Experience Estimation for WebRTC-based Video Streaming. In *European Wireless 2018; 24th European Wireless Conference*. [VDE Verlag GmbH], [Berlin], 2018.

[25] Matthew Wolenetz, Jerry Smith, Mark Watson, Aaron Colwell, and Adrian Bateman. Media Source Extensions: W3C Candidate Recommendation 12 November 2015, 2015.

Biographies

István Koren is a research assistant at RWTH Aachen University. Previous academic stations included i.a. TU Dresden and PUC Rio de Janeiro. He is working in the "advanced community information systems" (ACIS) at the information systems chair, RWTH Aachen University. István has worked in several large-scale European research projects in the area of Technology Enhanced Learning (Learning Layers, WEKIT, AR-FOR-EU). His research interests are Web engineering, Internet of Things and social computing.

Ralf Klamma holds diploma, doctoral and habilitation degrees in computer science from RWTH Aachen University. He leads the research group "advanced community information systems" (ACIS) at the information systems chair, RWTH Aachen University. He is known for his work in major EU projects for Technology Enhanced Learning (PROLEARN, GALA, ROLE, Learning Layers, TELMAP, Tellnet, CUELC, SAGE, BOOST, VIRTUS and WEKIT). Ralf organized doctoral summer schools & conferences in Technology Enhanced Learning, Web Engineering and Social Network Analysis. He published more than 200 scientific papers. He is on the editorial board of Social Network Analysis and Mining (SNAM) and IxD&A. His research interests are community information systems, serious games, augmented reality, web engineering, social network analysis, requirements engineering and technology enhanced learning.