Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

BOPlish – Distributed Content Communities Between Browsers based on WebRTC

Christian Vogt, Max Jonas Werner

Master Thesis

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Vogt, Max Jonas Werner

BOPlish – Distributed Content Communities Between Browsers based on WebRTC

Master Thesis submitted in the context of Masterprüfung

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

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Submitted on: 12 December 2014

Christian Vogt, Max Jonas Werner

Thema der Arbeit
BOPlish – Distributed Content Communities Between Browsers based on WebRTC

Stichworte
WebRTC, P2P, ALM

Kurzzusammenfassung
Anwender, die Inhalte im Web veröffentlichen wollen, müssen entweder eigene Server aufsetzen oder auf fremde Infrastruktur zurückgreifen. Die zunehmende Nutzung des Webs als Content-Sharing-Plattform verlangt jedoch nach Lösungen, die den Nachteilen zentralisierter oder restriktiver Plattformen entgegentreten. Seit einiger Zeit arbeiten die IETF und das W3C gemeinsam am WebRTC-Standard, um eine direkte Browser-zu-Browser-Kommunikation zu ermöglichen. Die vorliegende Arbeit präsentiert Browser-based Open Publishing (BOPlish), eine Infrastruktur-unabhängige Namens- und Inhaltezugriffs-Architektur für das Content-sharing in User Networks. Wir zeigen, wie BOPlish WebRTC verwendet, um eine einfach zu verwendende, sichere Veröffentlichungslösung für Inhalte zu ermöglichen. Ein eigenes URI-Schema dient als Lokations-unabhängiger Adressierungsmechanismus, um das Veröffentlichen und Beziehen von Inhalten von der zugrundeliegenden Infrastruktur zu trennen.
Christian Vogt, Max Jonas Werner

Title of the paper
BOPlish – Distributed Content Communities Between Browsers based on WebRTC

Keywords
WebRTC, P2P, ALM

Abstract
Users eager to publish content on the Web need to either set up a server or use third-party infrastructure. However, the increasing desire to use the Web as a sharing platform for content demands solutions that counter disadvantages of centralized or restricted platforms. Recent efforts are underway at the IETF and W3C to standardize WebRTC for direct browser-to-browser communication. This paper introduces Browser-based Open Publishing (BOPlish), an infrastructure-independent naming and content access architecture for sharing information in User Networks. We demonstrate how BOPlish leverages WebRTC for an easy to use, secure content publishing solution. A custom URI scheme serves as a location-independent addressing mechanism to separate publishing and content retrieval from the underlying infrastructure.

Contents

List of Figures

Listings

1 Introduction

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 among each other. Currently, perceiving user-generated content on the Web follows the client/server paradigm. Examples for such central content sharing community platforms are Facebook, Flickr and Youtube.

WebRTC is a new technology that enables Web applications to establish direct connections and data transmission between two browsers. This ressembles a major paradigmic change in the client/server dominated world of the current Web. Browser vendors such as Mozilla and Google already ship working implementations of the current specification status and a further deployment of WebRTC-enabled browsers from other vendors can be expected shortly.

Information-centric Networking (ICN) describes the idea of moving from a host-centric to a data-centric networking paradigm. It abstracts publishing and accessing content from the underlying infrastructure facilitated by a location-independent naming scheme. ICN potentially fosters the decoupling of user-generated publishing from a dedicated distribution system.

In this work, we introduce a decentralized, name-based publishing architecture called Browser-based Open Publishing (BOPlish) that pursues a similar objective. A BOPlish application runs in the Web browser and connects participating peers directly via WebRTC Data Channels, forming a virtual content-centric infrastructure where each user can publish and retrieve content. The system does not rely on additional external infrastructure and naturally prevents common privacy issues present in centralized architectures.

Similar to ICN, the accessed content is addressed employing a user-centric naming scheme decoupled from the delivering host. Our approach enables Web developers to plug-in custom application protocols that easily integrate into the BOPlish architecture. WebRTC acts as an enabler for BOPlish. Any device that has a WebRTC-enabled browser installed can join the overlay without requiring additional software. Encryption and secure transport of arbitrary data is directly provided by WebRTC.

This paper is structured as follows. We start our investigation with a wide range of back-

ground and related work that accumulated over the project lifespan in Sec. 2 before introducing our concept in Sec. 3. In Sec. 4, we describe the implementation of the name resolution mechanism used to resolve location-independent names based on the BOPlish URI scheme. Moreover, a user-facing API allows for easy development of applications running on top of BOPlish while hiding the complexity of the P2P system underneath. To verify our concept, we introduce demo applications and an emulation component to evaluate the core BOPlish system in Sec. 5. We continue by implementing an Application Layer Multicast (ALM) system in BOPlish (Sec. 6) and showcase how we include Flow Control in Sec. 7 before concluding in Sec. 8.

2 Background and Related Work

In this section, we introduce the fundamentals that our research is based on. Sec. 2.1 gives an overview of different P2P networking concepts while Sec. 2.2 outlines current Web technology achievements. In Sec. 2.3 we introduce WebRTC and the underlying technologies. We conclude this chapter with an introduction to Information-centric Networking (ICN) in Sec. 2.4 and a selection of related research activities in Sec. 2.5.

2.1 Peer-to-peer Networking

To define a Peer-to-Peer (P2P) system, thinking about the meaning of the word *peer* is helpful. In its original meaning, a peer is a person that is of the same rank or standing as another peer. In a computer-based P2P system, peers refer to the participating nodes in the system. The autonomous peers aim for a shared usage of distributed resources like computing power or bandwidth. Central entities are avoided and the system should be capable of self-organization. This implies that, instead of relying on a central entity as in a client-server system, every node acts as client and server to add its resources to the system.

Many Peer-To-Peer algorithms for Internet-based applications were originally designed for file sharing applications. As of now, the algorithms are used in many systems that depend on distributed resources like Content Distribution Networks (CDNs) or Internet telephony. [86, pp. 35–56] identified three main requirements of future Internet-based applications:

- *Scalability* allows a system to scale by several orders of magnitude without the loss of efficiency by eliminating bottlenecks caused by the systems design.
- *Security* and *Reliability* are of high importance for Internet-based applications where they are facing DDoS[1] attacks and consumer frustration when not constantly available.
- *Flexibility* and *Quality of Service* requirements form core criteria for modern Internet-based applications to accommodate features they were not conceived for.

[1] Distributed Denial of Service (DDoS) attacks aim at rendering a remote service unusable by overloading it

As of today, most services on the Web are based on the classic client-server paradigm which can currently handle the above-mentioned requirements sufficiently. However, vast amounts of centralized resources are needed. Thus, prominent services on the Web are typically operated by big companies. P2P systems try to provide alternative solutions and aim at avoiding the imposed centralization. P2P systems are typically classified as belonging to one of two main categories (see Fig. 2.1): *Unstructured* and *structured* approaches.

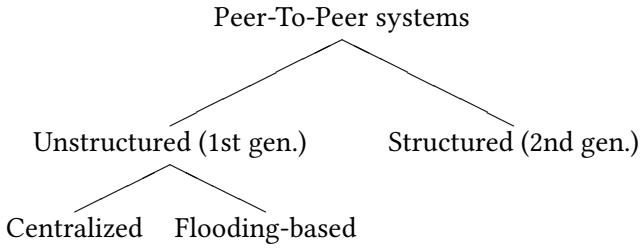


Figure 2.1: Overview of first/second generation Peer-To-Peer systems [86, pp. 35–56]

Napster[2] was one of the first applications that employed a large-scale centralized P2P system. These systems use a server-based, central entity for lookup and management purposes. The server acts as an indexer that keeps track of the available resources at all participating peers. When a peer issues a lookup to the server, it responds with a pointer to a peer (or a list of peers) that holds the requested content. The requester can then download the content directly from the other peer without involving the indexer. Lookups are simple requests to the server ($\mathcal{O}(1)$) which has to keep track of all participating peers. This results in increasing state complexity at the indexer when the network grows ($\mathcal{O}(N)$, where N is the number of participating peers). If the central entity fails due to attacks, overload etc., the whole system is rendered unusable. Moreover, it can be targeted by authorities to shut the system down (as happened to Napster).

Flooding-based approaches (e.g., early versions of Gnutella[3]) are completely decentralized as they do not rely on any central entity. State information is distributed in a mesh built from the participating peers. Every peer only knows about its neighbors in the mesh ($\mathcal{O}(1)$). This allows for a high fault-tolerance of the network as peer outtakes only have a small impact. Instead of querying a central entity, lookup requests are sent to neighboring peers which relay them to their neighboring peers (flooding). Apparently, this becomes increasingly inefficient for growing networks due to the high communication overhead every lookup generates ($\geq \mathcal{O}(N^2)$). As many requests can easily overload the system, the maximum hop count is typically restricted. Lookups are not thus not guaranteed to succeed when the content

[2] `http://www.napster.com/`
[3] `http://rfc-gnutella.sourceforge.net/developer/stable/index.html`

is rare among the participating peers.

Another form of P2P systems are hybrid P2P systems which combine classic client/server and the introduced P2P paradigms. Typically, the architecture is centered around a server-based system that additionally acts as a mediator. Peers get to know each other with the help of the mediator and form small-scale mesh networks. Content can then be accessed via both, the server and the neighboring peers. Such systems are used by, e.g., Akamai, one of the largest CDN providers via the NetSession Interface [62]. The software uses home computers as caching peers to aid in distributing content. Spotify, a large music streaming service, uses a similar approach to aid in delivering music to their customers [55]. Hybrid P2P systems are not decentralized as they rely on a mediator but offer promising performance enhancements and cost reduction for the service provider due to the reduced server load.

System	Per Node State	Communication Overhead
Centralized Server	$\mathcal{O}(N)$	$\mathcal{O}(1)$
Flooding-based	$\mathcal{O}(1)$	$\geq \mathcal{O}(N^2)$
Distributed Hash Table	$\mathcal{O}(log\ N)$	$\mathcal{O}(log\ N)$

Table 2.1: Complexity comparison of the different P2P approaches [86, pp. 79–93]

The above-mentioned approaches have in common that they do not employ a structured relation between the participating peers (the peers are *unstructured*). Both systems suffer from complexity issues, either communication overhead or the required node state information. Structured P2P systems aim at providing a scalable solution in between huge communication overhead and exploding node state information.

2.1.1 Structured P2P Systems

The idea of decentralized, self-organizing applications that do not rely on central entities sparked interest in the research community and the second generation P2P systems were developed in response to that. Such systems are structured in the sense that the participating peers are arranged according to some identifier space. Data that is to be stored in the system is then mapped to the same identifier space and assigned to a specific peer using a mathematical relation. The resulting topology can be efficiently queried and ensures that even rare content is guaranteed to be found. Logarithmic behavior can be assured for communication overhead and node state complexity [86, pp. 79–93].

Prominent applications incorporating a structured P2P system depend on Distributed Hash Tables (DHTs). In a DHT, queries are directed to a specific piece of content (i.e., the content's

key) instead of a location. They offer an interface similar to traditional hash tables. This abstraction makes it easy for the developer to interact with the P2P system without in-depth knowledge about the underlying P2P protocol. While being transparent to the application, the DHT-layer is responsible for efficiently routing the request to the corresponding peer. Figure 2.2 illustrates the DHT interface. To store a value, the `put()` operation is used in conjunction with a hash function that calculates the corresponding key for the value. This value is stored at a corresponding peer that is in charge for a specific key space. The `get()` operation is used to look up a value.

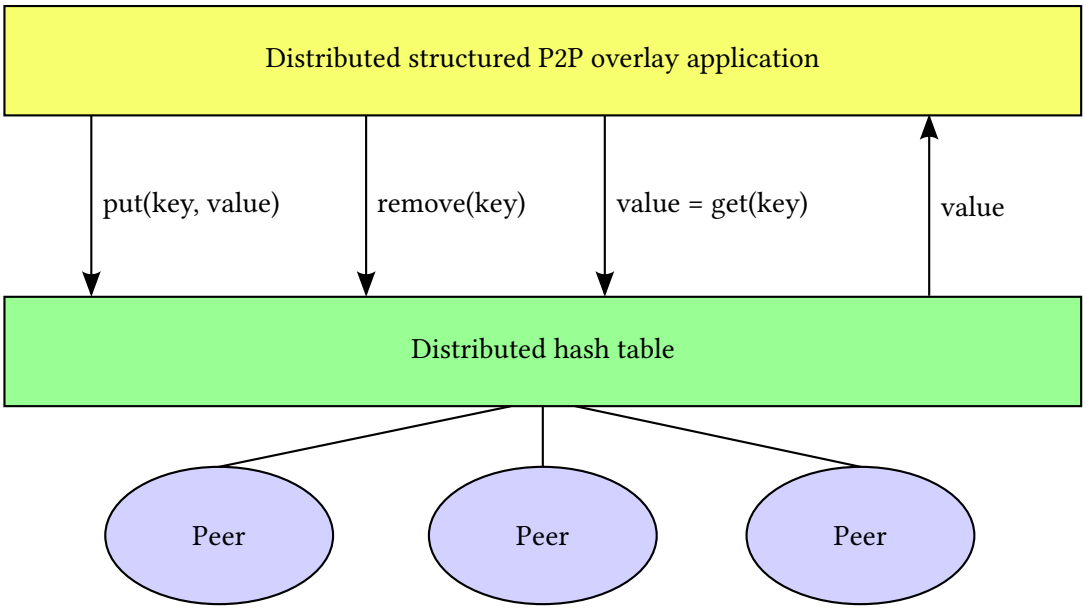


Figure 2.2: Application interface for structured DHT-based Peer-To-Peer overlay systems [86, pp. 79–93]

A DHT maps participating peers onto a large uniform identifier space. Content is then mapped to the same identifier space by consistently assigning keys to each data object. Every peer is assigned a range of content identifiers for which it is responsible, according to the DHT protocol. Peers maintain a routing table consisting of selected peer identifiers. When a traversing lookup arrives, the protocol chooses a peer numerical closer to the requested key out of its routing table to send the request to. This is possible because of the consistent mapping of content and peers to the same identifier space and the resulting mathematical relation between them.

DHT-based systems can make assurances about the number of overlay hops it takes to reach the peer that holds the content, typically $\mathcal{O}(\log\ N)$. Such lookups are guaranteed to succeed

if the requested key exists in the system. Moreover, every peer only has to store $\mathcal{O}(\log\ N)$ states. This combination of properties allow DHTs to scale to extremely large numbers of peers [86, pp. 79–93].

Besides the usage as a basis for structured overlays, so-called one-hop DHTs also find a use in other systems such as the Amazon Dynamo NoSQL-Database [24], or the GlusterFS distributed filesystem[4]. These system scale with $\mathcal{O}(1)$ at the cost of greatly increased maintenance traffic as every peer maintains a routing table that contains all other peers in the system. Thus, they are only applicable for reliable, server-based systems or very small groups.

2.1.2 DHT Design Challenges

DHT implementations are typically based on geometries like rings (e.g., Chord [89]), trees (e.g., Pastry [79]) or tori (e.g., Content Addressable Networks [70]). The difference between the approaches are the search and management strategies, as well as topology-based enhancements for routing decisions [57].

DHTs leverage the flat identifier space to structure the participating peers. As an example, the Chord algorithm uses a ring structure where the numeric identifiers are placed upon. Every peer knows the peers that are numerically closest to him in each direction of the ring. A specific content range is associated with every peer ranging from the peer's ID to the predecessor's ID. In the event of peers joining or leaving, the associated identifier space has to be changed accordingly. This is achieved by triggering a repair mechanism that maintains the list of connected peers. The keys associated to the departing client are lost; as such applications building on top of a DHT typically use replication to spread the keys throughout the system.

A problem that occurs when mapping peers to an uncorrelated overlay is that the overlay topology does not reflect the real topology (see Fig. 2.3). This can lead to long network routes for overlay hops, commonly expressed as the Delay Stretch. Delay Stretch is a metric that is defined as the ratio between the resulting path and a unicast-like communication between these peers using a distance metric like round-trip delay. It is possible to lower the Delay Stretch using Proximity Neighbor Selection (PNS). PNS enables a DHT peer to influence the choice of peers in its peer table. A search algorithm is used to find nearby peers whereas the crucial part of this algorithm is an efficient latency prediction between peers that only consumes minor peer resources (such as Vivaldi [21]). PNS can significantly reduce delay stretch.

When the overlay topology has been established, lookups can be issued against the DHT. Such lookups are directed against a specific piece of data using its identifier. The DHT transparently

[4] `http://www.gluster.org/`

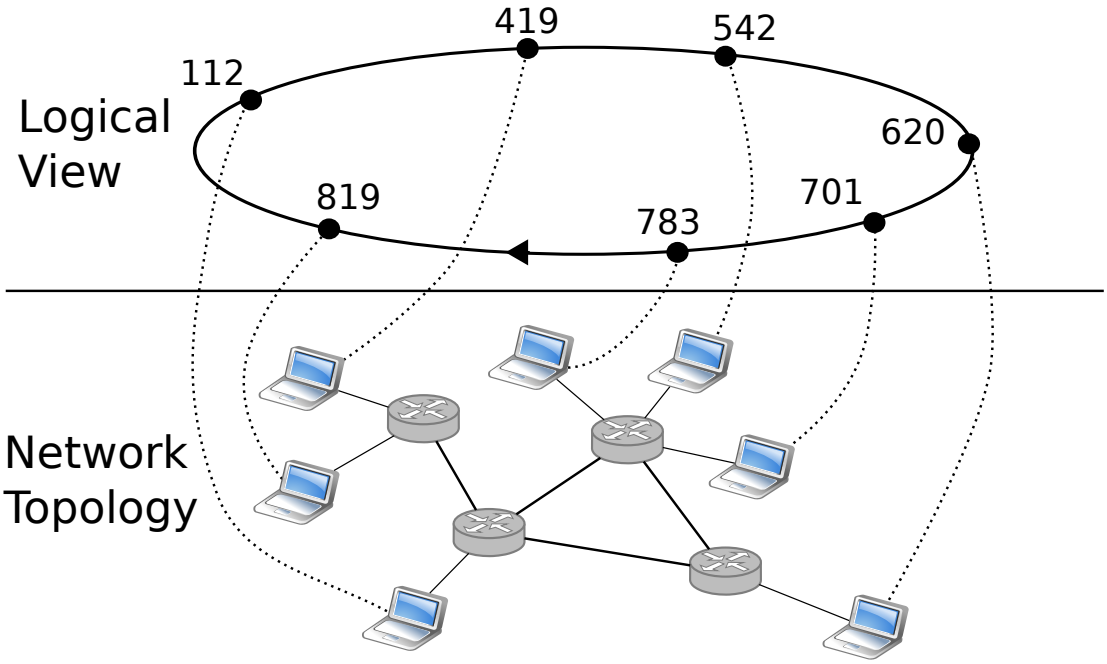


Figure 2.3: The mapping between overlay and underlay topology may lead to disadvantageous paths [86, pp. 79–93]

routes the request to a corresponding peer (this is called Content-based Routing). In Chord [89], this can be achieved by traversing the ring topology until the associated peer is reached. While this approach is simple to implement it would take an average $\mathcal{O}(\frac{1}{2}N)$ steps to reach the peer. Chord therefore maintains a routing table that stores links to other neighbors to shortcut the route. The distance to the neighbors exponentially increases such that the ith routing table entry points to successor $n + 2^i$. This allows any lookup to finish in a maximum of $\mathcal{O}(log\ N)$ steps.

Data storage is managed by the application building on top of the DHT. The application might not store actual data but use a separate layer of indirection, i.e., the values might only consist of a link to the actual data (such as a URL). Generally speaking, integrating lookup and storage functionality in the DHT benefits latency because the data can be piggybacked on the fetch response instead of triggering a new request to the indirection layer.

Applications that store data have to decide on the size of the data units. A data unit might refer to a disk sector-like fragment of a file, a whole file or even an entire file system image. In general, large unit sizes lead to a inferior number of lookups while splitting large files into smaller units distributes the load over more peers.

2.2 Browser APIs and Networking Functionality

In order to transfer the principles of P2P networks to web applications, developers have to leverage at least one of the networking technologies that browsers offer. Additionally, the understanding of fundamental concepts of the Web is key to making applications responsive and provide a decent user experience. Those concepts can be broken down into three parts: Identification of resources via Uniform Resource Identifiers (URIs) [8]; transfer of the application via Hypertext Transfer Protocol (HTTP) [32] and presentation of information using Hypertext Markup Language (HTML) [46]. Besides the named three standards further technologies exist such as CSS (for layout/design), JavaScript (for programmatic interaction with documents), HTTPS (for secure transfer using HTTP over SSL/TLS) and the Document Object Model (DOM [48], used to interact with the presentation programmatically, e.g., via JavaScript).

These technologies gained importance over the last 15 years. Until the beginning of the 21st century, Web pages were mostly static (or dynamically rendered on the server) and users could barely interact with the content. New technologies like the XMLHttpRequest object and improved JavaScript performance in the browsers served the transformation of Web pages into Web applications that could be used interactively (see Fig. 2.4); applications such as Google Maps gained popularity. Certain workarounds for enabling push events from server to client like HTTP long-polling[5] were introduced for real-time use cases such as Web chats. Standardization efforts of such workarounds have spawned three major networking technologies that programmers can leverage to build highly interactive real-time applications: XMLHttpRequest, WebSocket and Server-sent Events. These are described in more detail in the following sections.

2.2.1 XMLHttpRequest

On simple web pages that use traditional HTTP, every user interaction – such as the submission of a form or the click of an anchor – results in a complete page reload. The XMLHttpRequest (XHR) object as specified in [96] makes it possible to asynchronously (i.e., in the background) open a connection to a remote server using JavaScript, without reloading the page, as outlined in Fig. 2.5. This technique is known as Ajax which initially stood for “Asynchronous JavaScript and XML” because in the beginnings of Ajax, it was used to transfer mainly XML documents; nowadays most applications transfer data using a more lightweight approach, e.g., JSON. The benefit of using Ajax is that certain actions conducted by the user (e.g., clicking a button) do not result in a full page refresh anymore. This way the overhead of retrieving new data from

[5] `http://en.wikipedia.org/wiki/Push_technology#Long_polling`

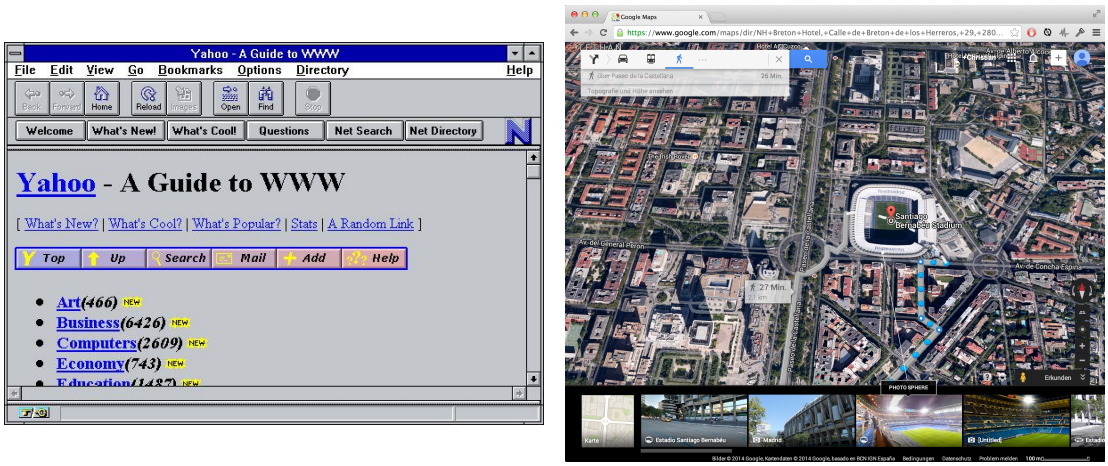


Figure 2.4: In the early days, the Web consisted of static pages containing mostly text, images and hyperlinks (left). Modern Web applications such as Google Maps (right) are highly interactive single-page applications.

the server is kept to a minimum and the Web application becomes more responsive.

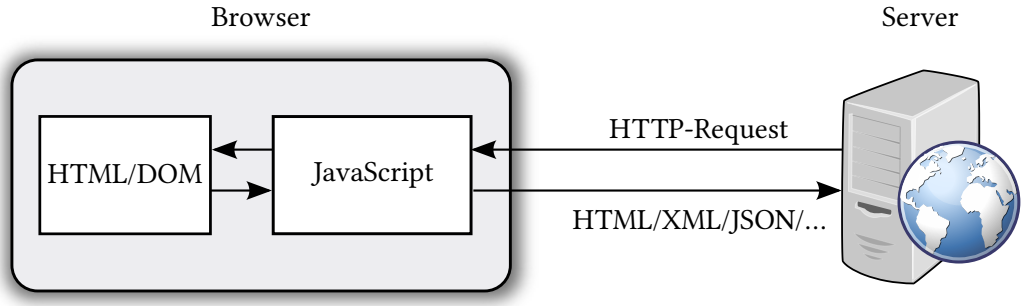


Figure 2.5: Schematic view of an Ajax request

2.2.2 WebSocket

The concept of WebSockets [44] extends the use cases enabled by XMLHttpRequest (unidirectional communication) with the possibility to establish a bidirectional channel between client and server, as depicted in Fig. 2.6. The WebSocket standard introduces two new URI schemes, `ws:` and `wss:`, for unencrypted and encrypted connections, respectively. The WebSocket protocol uses the HTTP Upgrade mechanism in the initial handshake (a simple HTTP GET-like request) to switch from HTTP to WebSocket. After a successful establishment, browser and server are capable of communicating in a bidirectional way. A main difference between Web-

Socket and XMLHttpRequest is that the former is not dependent on HTTP. Rather, WebSocket enables a developer to run a custom protocol on top of a WebSocket. In a way, this is the equivalent of a raw network socket, just in the browser.

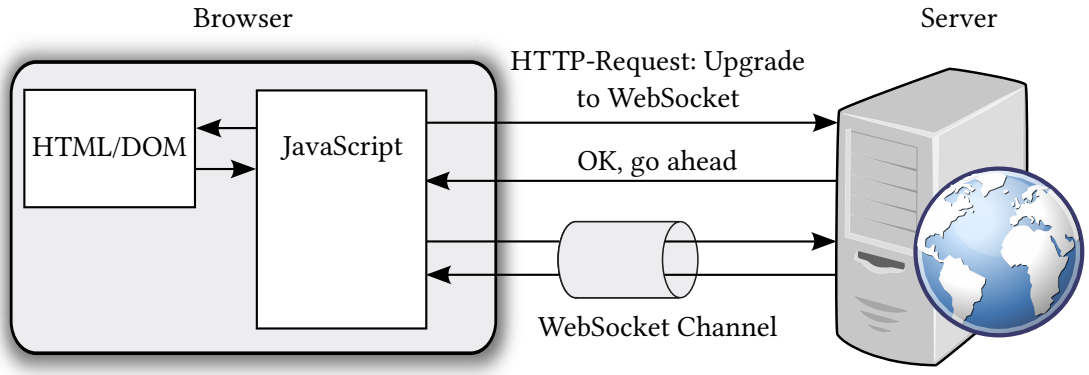


Figure 2.6: Schematic view of a WebSocket connection

2.2.3 Server-sent Events

The third networking mechanism mentioned here is used to push data from the server to the client. For this, the W3C specification [43] adds an additional DOM interface to browsers: EventSource. A programmer may instantiate an EventSource object providing a URL. The browser opens a connection to the URL (by adhering to the same-origin policy) that is held open. This way the server may push data to the client when it becomes available. Server-sent Events (SSE) qualify as a lightweight alternative to WebSockets while enabling similar use cases such as a push-based updating of news feeds. Fig. 2.7 outlines the mechanism of Server-sent Events.

2.3 WebRTC

WebRTC is a standardized protocol suite that enables two endpoints to communicate directly over a UDP-channel, paired with a JavaScript API for Web applications [7]. It enables real-time audio, video and data sharing between two endpoints. Instead of relying on third-party plug-ins, WebRTC is supposed to be built into Web browsers, thus instantly enabling a huge user base while keeping a simple-to-use API. Under the hood, though, WebRTC includes a multitude of functionality previously not available to Web browsers. This includes a signaling

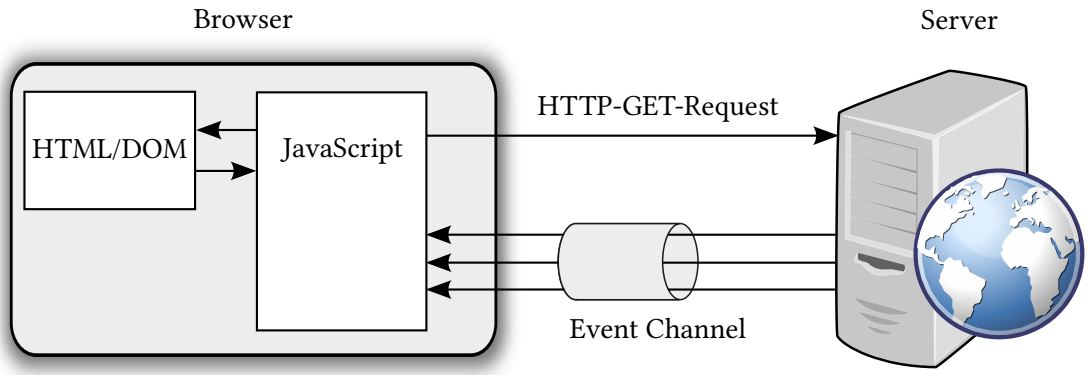


Figure 2.7: Schematic view of a Server-sent Event connection

infrastructure, connection negotiation functionality and the implementation of many new protocols that are required to match the new requirements which we are going to discuss now.

Opposed to all other browser technologies (XHR, WebSocket etc.), WebRTC tunnels all data over UDP (except for signaling messages), running custom application-level transport protocols through this tunnel. Despite all the new functionality it introduces, WebRTC is also limited in the way that it allows two browsers to interconnect and exchange data. The standards neither include topology- nor routing-related topics. Currently, the specification of WebRTC is in active development and the API as well as the underlying protocols are still in heavy flux. It is not yet clear which functionalities (besides audio, video and data channels) will be included in the final specification (e.g., real-time text). This section details how the different protocols work together to cope with real-time communication, connection establishment and signaling requirements.

2.3.1 API Overview

A typical WebRTC session (as shown in Fig. 2.8) starts with a user visiting a website by entering a URL into her Web browser. The URL contains the server to connect to (i.e., the DNS name). The requested server delivers the application (consisting of HTML, CSS and JavaScript code as well as resources such as images) to the client's browser using traditional HTTP. The browser now initiates a WebRTC connection triggered by executing JavaScript code received from the server. In Fig. 2.8, the server also handles the signaling of a WebRTC connection between two peers, serving as a central connection establishment entity. While this is a typical scenario, using any arbitrary channel is also possible. When the signaling is done, any further communication is handled merely by the browsers.

Three main interfaces are exposed to WebRTC-enabled Web applications. That is, the

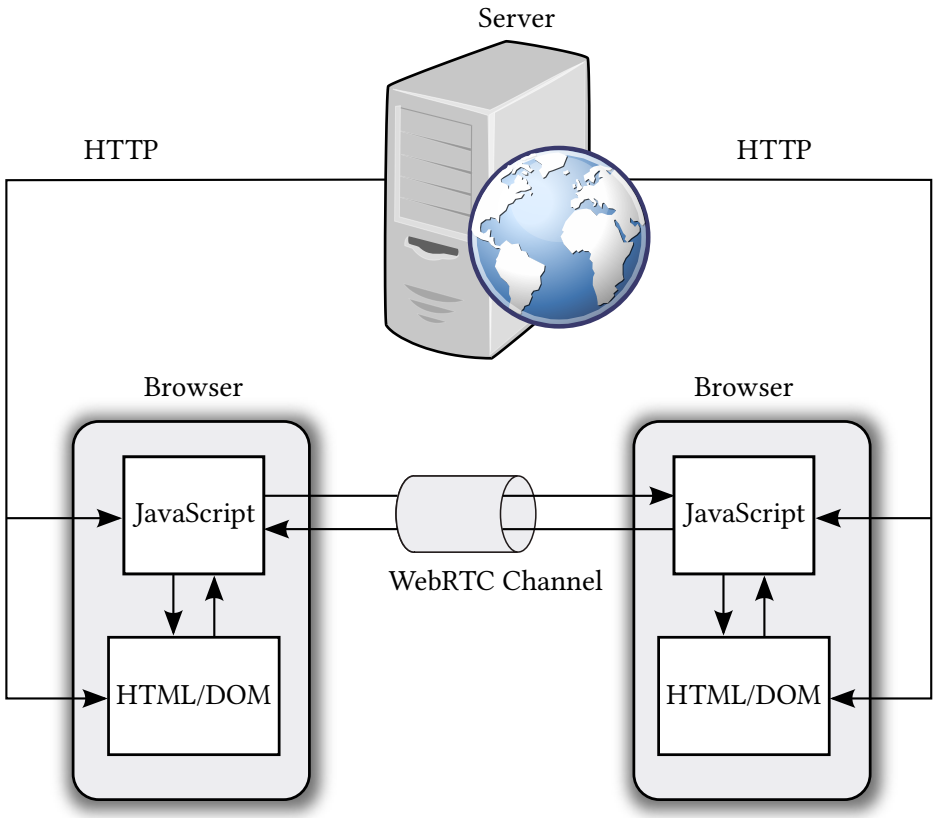


Figure 2.8: Schematic view of a WebRTC connection

`RTCPeerConnection`, `getUserMedia` and the `RTCDataChannel`. The first one is responsible for connection establishment and maintenance using the Interactive Connectivity Establishment [75] (ICE) mechanism. Moreover, it provides connection status information and, despite the complexity of WebRTC, serves as a single entry point for the application. `getUserMedia` is used to gain access to audio/video devices on the peer while the `RTCDataChannel` interface enables exchanging arbitrary data between endpoints.

2.3.2 Entities in a WebRTC-based System

We'll now focus on the different participants acting in WebRTC communication. As a starting point, Fig. 2.9 shows a stacked view of the complete protocol suite that is investigated in this section.

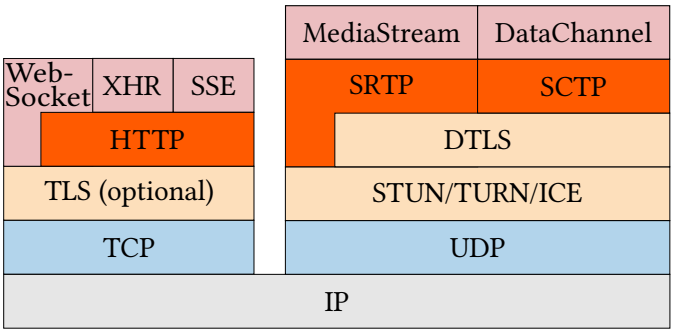


Figure 2.9: The protocol stack of WebRTC consists of a connection management component for establishing and maintaining connections, an A/V component and a Data Channel component. Communication is always encrypted end-to-end [36].

Server

The server as outlined in Fig. 2.8 has two purposes: Deliver the application that makes use of the WebRTC JavaScript API and enable signaling between two browsers. Technically, these two purposes may be handled by two different servers, while typically both functions are conducted by one server entity.

Browser

The browser acts as the runtime environment of all WebRTC code and therefore is a critical component in the WebRTC infrastructure. To enable use cases such as an audio/video conference, the WebRTC application must be granted access to a microphone and/or camera attached to the user's computer (via `getUserMedia`).

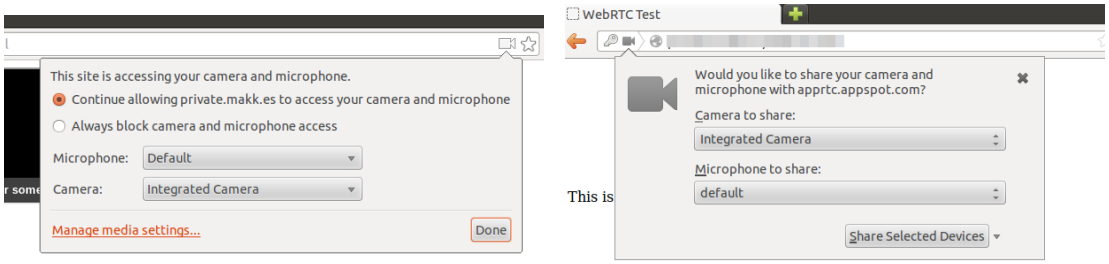


Figure 2.10: When a web application indicates the desire to access the computer's camera and/or microphone, browser's must ask the user for consent. The consent prompts shown here are those of Chrome (left) and Firefox (right).

The specification states that "allowing arbitrary sites to initiate calls violates the core Web

security guarantee; without some access restrictions on local devices, any malicious site could simply bug a user." [72] Because of this, browsers are obliged by the specification to "obtain explicit user consent prior to providing access to the camera and/or microphone". Fig. 2.10 shows two implementations of such a consent prompt; one for Chrome and one for Firefox.

Apart from the consent mechanism, the user must have the means of determining that a call is in progress. Current browser implementations use indicators as show in Fig. 2.11. A topic of recent discussion among browser developers is that of usability and general user experience with these consent mechanisms on mobile devices. There currently seems to be no final agreement on best practices, e.g., what the browser should do when it is sent to the background on a mobile phone (options discussed right now are to completely stop sending data or to indicate hardware access in a notification area of the phone).

Both mechanisms – consent, "call in progress" indicator – are currently only specified for media streams and not for Data Channels. The latter can be initiated by an application at will and without the user noticing. In future versions of the WebRTC specification, though, it may be possible that such mechanisms are established for Data Channels, too.

Figure 2.11: Browsers show an indicator of an on-going call so that the user knows of a possible A/V transfer to another peer. Chrome (left) puts this indicator on the top of the tab, Firefox (right) shows it in the address bar and permanently in the menu bar.

Signaling Path

Connecting two peers using the WebRTC handshake involves a negotiation sequence [93] as shown in Fig. 2.12. WebRTC uses the SDP Offer/Answer described in [77]. It allows the peers to learn about each other (e.g., NAT traversal options, supported codecs) and agree on parameters for the connection (e.g., IP/port combination). Lst. 2.1 shows a Session Description Protocol [41] (SDP) message as created by the connection initiator (Alice) by calling `createOffer()`. The offer is then sent to the remote peer using the arbitrary signaling channel (`sendOffer`). The callee (Bob) processes the offer, creates the answer message and sends it back over the signaling channel (`sendAnswer`). Alice then uses the information contained in the SDP to initiate the `PeerConnection`, therefore finalizing the handshake procedure. Signaling in

WebRTC is done out-of-band, meaning that every application has to provide its own signaling channel. Common methods are XHR, SSE and WebSockets, which come in both, encrypted and unencrypted variants.

```
1 v=0 // protocol version
2 o=Mozilla-SIPUA-33.0 14557 0 IN IP4 0.0.0.0 // origin
     identifier
3 s=SIP Call // session name
4 t=0 0 // may indicate start/stop times; not used in WebRTC
5 a=ice-ufrag:c43d936f // ICE parameter
6 a=ice-pwd:0a0435ce7407bfc0ec43a953278916c7 // ICE parameter
7 a=fingerprint:sha-256 <omitted> // DTLS parameter
8 m=application 49687 DTLS/SCTP 5000 // Request for Data Channel
9 c=IN IP4 84.130.199.184 // Connection Endpoint (overwritten by
      ICE)
10 a=sctpmap:5000 webrtc-datachannel 16 // Data Channel parameter
11 a=setup:actpass // Identifies the offerer, wait for answer
12 // ICE candidates following:
13 a=candidate:0 1 UDP 2130379007 192.168.0.20 49687 typ host
14 a=candidate:1 1 UDP 1694236671 84.130.199.184 49687 typ srflx
     raddr 192.168.0.20 rport 49687
```

Listing 2.1: SDP offer message (created by a caller) used for a WebRTC Data Channel session negotiation.

As the SDP information is a simple string (Lst. 2.1), it can easily be transmitted over any signaling channel as a text blob or transcoded to other formats (e.g., XML). The string consists of multiple lines, each starting with a single case-sensitive character that describes the attribute usage [41]. Notable properties are "m="-lines which specify the different media endpoints. In Lst. 2.1, a single Data Channel is requested by the caller. Moreover, "a="-lines specify extensions to the original SDP specification. WebRTC uses the ICE mechanism to allow NAT traversal between two endpoints. An ICE agent gathers tuples of IP/port combinations (candidates) by querying the operating system as well as an external Session Traversal Utilities for NAT [76] (STUN) server. This allows to gather both, local and public IP addresses even in the case of a NAT environments. If explicitly configured, ICE can also use a Traversal Using Relays around NAT [66] (TURN) server which acts as a proxy when all other attempts fail (as in the case of both peers being behind symmetrical NATs).

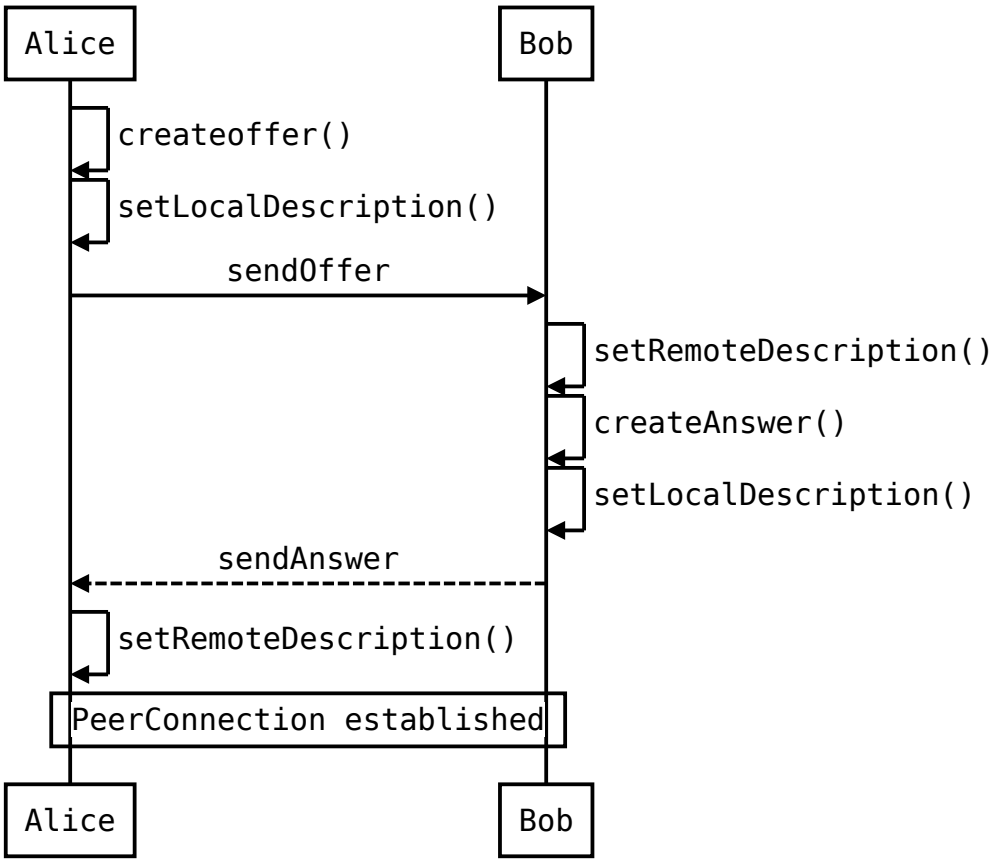


Figure 2.12: Alice establishes a WebRTC connection to Bob by following the JSEP signaling sequence using an arbitrary channel to transmit the offer/answer messages.

Media Path

After two endpoints successfully finished the handshake procedure as described above, content can be transferred. WebRTC distinguishes between media and other arbitrary data content. Media is sent using Secure Real-time Transport Protocol [5] (SRTP) while Stream Control Transmission Protocol [87] (SCTP) is used for any other data. The solution concept described in this thesis is built upon Data Channels, which are further elaborated on here. More information on the media part of WebRTC can be found in the corresponding drafts ([4] contains an overview of all protocols used by WebRTC implementations). To describe the WebRTC Data Channel implementation, the requirements specified by the IETF in [53] serve as a starting point:

- Support for multiple, simultaneous data channels ①
  - Reliable and unreliable transmission modes for each channel ②

  - In-order and out-of-order message delivery for each channel ③
  - Support for prioritization between channels ④
- Message-oriented API ⑤
- Support for congestion and flow control mechanisms ⑥
- Security considerations (confidentiality, integrity and source authentication) ⑦

To satisfy all these requirements, the IETF agreed on a layering of Data Channels over SCTP over Datagram Transport Layer Security [71] (DTLS) over ICE over UDP. SCTP was designed as a message-oriented transport layer protocol but can also be used on top of DTLS as described in [92]. SCTP features both, reliable and unreliable transmission of messages and can be configured for ordered or unordered delivery. An SCTP connection between two participants is called an association which can carry multiple independent streams at once while only consuming a single port at either side of the connection. Streams are unidirectional channels between endpoints that transport the actual messages (thus, a Data Channel consists of two streams, one in either direction). By configuring the message properties, SCTP can thus satisfy ①, ②, ③, ⑤ and ⑥. DTLS provides the mechanisms for the security requirements (⑦). Inter-channel prioritization (④) is implemented in the Data Channel layer itself (i.e., by the browser). Because most browsers are expected to operate behind a NAT, ICE is natively provided to aid during connection establishment. ICE uses STUN and its extension TURN to circumvent connectivity problems in NAT environments.

2.4 Information-centric Networking

The Internet revolution started after the World Wide Web had introduced a uniform, simple architecture of separating content publication and provisioning from content retrieval. The decoupling of publishing information from its consumption in space and time is a core element of the rich publish/subscribe paradigm [30]. In recent years, (proprietary) Content Delivery Networks (CDNs) have shifted this server-centric approach to the network that mirrors one-to-many communication for which the initial Internet architecture has not been built [40].

The ideas of Information-centric Networking (ICN) have taken up the well-established CDN concept of in-network storage and replication towards end-user communities, while adding the core objective of an open future Internet design [105]. The latter requires resolution of the three major challenges: naming, security and routing [56]. In ICN, the underlying network layer must be capable of directing a named data request to a location completely transparent to the

requesting client, and it must provide an independent verification of the supplied content. As a result the location of data becomes irrelevant, making it simple to introduce caches distributed throughout the network. Many such architectures have been introduced, prominent examples being TRIAD [37], DONA [54] and NDN [50]. These and further solutions differ in naming, security, and routing, but all show a high interdependency among these three [35].

Unlike the Web URL-scheme, ICN uses names that are independent of a location or server instance. An explicit identifier is assigned to each data object. Two competing approaches exist, hierarchical and flat (e.g., hashed) naming. ICN implementations like DONA and NDN use flavors derived from either of the two approaches (Lst. 2.2):

```
1 ndn://alice/images/image.png
2 dona://134(...)0f6:dfe(...)164
```

Listing 2.2: Example for hierarchical identifiers (NDN) and flat identifiers (DONA)

Hierarchical names have the benefit that they can be aggregated, provided name prefixes and content locations coincide. When routing on the names itself, routing table sizes can be reduced by aggregating names. NDN uses such hierarchical names. Aggregation could be performed at the ISP level (with ISPs assigning prefixes to their customers), but this reintroduces a binding to location. The existence of the location-identity binding is the main argument for flat names (as used in DONA), which allow for a complete decoupling of location and identity but cannot easily be aggregated. Coping with a huge amount of unaggregated identifiers requires either huge routing tables or external infrastructure. As a reference, the Google index holds $\mathcal{O}(10^{12})$ entries which is a lower bound for any ICN approach. For comparison, DNS ($\mathcal{O}(10^{8})$) and BPG ($\mathcal{O}(10^{5})$) not only cope with much lower amounts of entries but also with a low update frequency compared to ICN (where every object publication results in a new entry). Finding a scheme that allows for both, effective aggregation and location-independence of the system without bloating routing tables is still subject to research activities [35].

Another aspect of the debate how to design content identifiers is the decision between human-readableness and cryptographic expressions for self-certification. DONA, for example, uses a cryptographic hash of the content as its identifier which can be used to verify the contents integrity. With NDN, this does not work as no natural linkage exists between identifier and content. To provide content verification, NDN requires an external trust mechanism as described in [85].

Each content request in ICN should be directed to a nearby surrogate in the network. When a location of the content is found, it has to be transferred to the requester. Different routing approaches exist to find a path over which the actual content is transferred. Depending on the ICN implementation, routing is performed directly on names (e.g., NDN) or decoupled by

an external resolution service (e.g., DONA). In a coupled approach, data forwarding follows a reverse path (Reverse Path Forwarding (RPF)) identified by the name resolution. In a decoupled approach, data is forwarded independently of content routing paths using regular IP/BGP routing [105].

Coupling the data routing means to either a) store routing states in the intermediate hops traveled by the name resolution query or b) integrate this information into the content query packets on the way. Decoupled approaches allow for more flexibility, as control and data flows can be separated. On an Internet scale, both approaches must be seen as a severe challenge [56].

2.5 Prior Work on the Thesis Subject

Our concept of user-centric content networks revolves around the idea that every participant of a community is able to name and publish content. All of the (static or dynamic) content a user wishes to publish is assigned a URI that is derived from the user's unique name. The concept of user-centric naming has been explored by other authors. In [2], Allman describes the concept of a "personal namespace". The author lays out several problems with current naming systems such as DNS and URLs: Names are location-bound as is the case with URLs, where the host name is resolved to a specific location on the network. Additionally, e.g., domain names are mentioned as ambiguous so that users do not actually know by the domain name who the owner of the domain might actually be. The author distinguishes three different parties that are involved in creating and accessing a name for a content item: the consumer, the content provider (e.g., a user who shares a file) as well as the service provider (e.g., Flickr or Facebook).

The "pnames" system proposed in [2] acts as an indirection between personal names assigned to a specific user and actual names like URLs or host aliases. This enables users to reference, e.g., Bob's e-mail address as `Bob:mail`. For sharing such pnames the author proposes the usage of a DHT to resolve the flat pname identifiers.

In a follow-up to "pnames" the authors provide the outline and a prototypical implementation of a more abstract idea that is based on the concept of storing and referencing meta data of arbitrary content [13]. That system is called Meta-Information Storage System (MISS). MISS is meant to be operated on a global scale at ISP level. All MISS servers are interconnected in a global DHT that is used to find the MISS server that holds a specific information item. The authors thus introduce a lookup layer for retrieving meta-information of content.

A high-level description of user-centric networking is presented in [58]. The authors start with the idea that each user in a specific interest group offers a set of services to the group. For

interconnecting users the authors propose to leverage existing social networks such as Twitter or Facebook and retrieve unique user identifiers from there. This way it is possible to leverage existing relationships between persons. A tuple of `(user name, service name)` is proposed to address services offered by a specific user. This makes it possible to decouple the service name from the host that offers the service while at the same time coupling the service with the user offering it (e.g., to ensure authenticity).

The IETF is currently working on an Internet Draft for a user-centric SIP (Session Initiation Protocol) approach [52] that is based on RELOAD specified in [51]. RELOAD defines a powerful framework for P2P storage and messaging, including a security model, NAT traversal and a pluggable topology mechanism (with a Chord variant as default topology plug-in). RELOAD is designed so that specific overlay applications are to be implemented on top of a RELOAD network. One such application is the SIP usage specified in [52]. This usage employs RELOAD to establish SIP sessions via the P2P overlay and defines a naming scheme, eventually defining a completely user-centric distributed telephony service. The RELOAD DHT stores a mapping from their AOR (e.g., `alice@dht.example.org`) to their node ID in the P2P network. This mapping is then used by other users to retrieve the node to connect to.

Research on leveraging native browser technologies – each achieving a different set of goals – is already being conducted: [60] 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. However, most current implementations and demos leveraging WebRTC are currently focusing on audio/video communication using SIP, like sipML5[6].

Many large-scale P2P systems have been deployed. Spotify, one of the leading music streaming services, uses a peer-assisted P2P system to distribute content between users of their desktop application and therefore reduce the load on servers [55]. Besides the unstructured, peer-assisted content distribution mechanism, they created a DHT overlay that employs the popular Publish/Subscribe communication paradigm. The DHT overlay allows users to exchange notifications and they claim to handle billions of publications per day [82].

Ownership of personal data in Web applications is a matter of ongoing passionate discussion. The main problem is that data resides on the providers' servers. A peer-to-peer architecture has the potential to mitigate the impacts of storing data on foreign servers since it can be distributed and encrypted. [38] investigates the possibilities of a censorship-resistant peer-to-peer collaboration architecture, but without focusing on Web technologies. [33] show a way to evade censorship by making every browser a proxy using WebSockets.

[6] `http://sipml5.org/`

3 Browser-based Open Publishing

Currently, publishing content on the Web requires access to infrastructure such as Web servers, whose names are coupled to the DNS system. The Web centers arounds hosts with perpetual connectivity to the Internet. If a Web servers is disconnected, all content – addressed via URLs – is inaccessible because its name is (via DNS) bound to the (now disconnected) host. ICN approaches this problem and elevates the meaning of content by assigning explicit identifiers to data rather than referring to the location of content, as is done with HTTP URLs. Content can be stored directly in the network, making the network itself content-aware. Eventually, the network provides functionality to store, cache and (to a limited degree) search for it. The ICN approach, however, suffers from conceptual shortcomings as Wählisch et al. [100] have pointed out. Besides unresolved security issues, control over the infrastructure is shifted towards end users. This paradigm opens up the control plane and requires modification of routing states at every user-generated publishing act.

In our contribution, we introduce a bottom-up approach that enables use cases similar to ICN but without replacing existing infrastructure. A content-centric network is established between end user's browsers and data is addressed using location-independent identifiers. We call this approach Browser-based Open Publishing (BOPlish). It focuses on distributing user-generated content within interest groups that we call User Communities. Web application providers could potentially benefit from such a system by reducing the server's bandwidth consumption and transfer delay. The clients in turn are not required to rely on a server for sharing content with other users. As we will show further on, BOPlish also enables efficient group communication through overlay multicast. As the browser is the natural application platform for the Web, we want to leverage its broad deployment and OS independence. It enables users to directly participate in a BOPlish community without installing additional software.

A User Community consists of a number of peers (typically browsers) that are connected to each other via a P2P network. A Web server delivering a BOPlish application serves as rendezvous component for joining one specific community. A user can join the P2P network and may close the connection to the Web server without losing any functionality provided by

BOPlish. Prior to joining a User Community, a peer has to acquire a unique peer ID (its overlay address) and the user has to authenticate at an identity provider. The combination of peer ID and username is then used to join the community and stored in a Distributed Hash Table (DHT) with the username as key and the peer ID as value. From then on, BOPlish provides a service for naming (using URIs), publishing and retrieving content between all participants.

On top of the described mechanisms, ensuring users' privacy is an important goal for BOPlish. After the revelation of world-wide spying programs like Bullrun or PRISM, privacy in Internet applications has become a mainstream topic. As a result, there is increasing demand for privacy by many Internet users. Classic Web applications based on client/server paradigms pose the risk of users exposing all their data (chat communication, documents, friendship relations) to a single entity like Facebook, Google or Apple. Thus, the increasing sensitivity of Web users cannot be satisfied by these applications, anymore. WebRTC transports, on the other hand, are always encrypted end-to-end and the browser-to-browser communication paradigm enables the creation of pure P2P Web applications. These two properties make it less likely for eavesdroppers with access to transport routes or community infrastructure (the P2P network) to reveal sensitive data.

This chapter proceeds with a description of use cases BOPlish should enable in Sec. 3.1. After gathering the use cases we continue with a description of the requirements that our solution shall fulfill in Sec. 3.2. Sec. 3.3 dives into the details of our solution concept and in Sec. 3.4 we close this chapter with a high-level description of the resulting architecture.

3.1 Use Cases

We now present use cases that are inspired either by the current Web or explicitly outlined for ICN (e.g., in [65]).

Document Sharing And Search Current file sharing applications can be divided into two main groups: Server-based and P2P-based. On the Web, file sharing is implemented using the server-based approach. The drawback here is the reliance on a centralized service or the requirement to setup a custom server. Moreover, users have to trust the service provider with regards to content integrity and privacy concerns. A user-centric approach would counter these disadvantages in the following ways: Users share documents directly from one browser to another. The publication of a document does not rely on setting up a Web server, uploading the document to a central instance or changing DNS entries. Similarly, it shall be possible for users to search for content on other peers.

To share a document, a user registers for an account, uploads a specific document from the filesystem to 'the Web' and grants either public access or to a group of collaborators. Typically, this is done by sharing an identifier via an external channel or by using a front end to invite other registered platform members.

Conversational Apps Real-time text or audio/video chats are of growing popularity. The increasing usage of social collaboration tools in the private as well as the enterprise context serve as a ground for conversational apps. On the Web, users employ centralized applications provided by service operators.

In a group chat context, all users of a community need to call and establish sessions. A single user must be able to open a chat room and invite other users (in some way connected to this user, e.g., via a friendship relation). The network transport must provide appropriate real-time services.

Group Collaboration and Gaming In group collaboration applications and multiplayer games users collaborate in self-defined groups. Such applications allow for the creation of groups, varying memberships and the invitation of participants. In addition, the application state (e.g., the position of players in a game or the content of documents in a groupware) must be transparently accessible for all members of a group.

Mobility and Offloading More and more people use Internet services from mobile devices like smartphones or tablets. Under mobility, the user network must be able to cope with frequent network address changes. Publishers can accomplish this goal by offloading content to (stationary) third parties that promise better connectivity. Consumers can partially mitigate rapid address changes by pre-fetching content.

Replication and Synchronization Users generally demand high content availability and fast access. Replication accomplishes this by storing multiple copies of the content (replicas) on different, independent systems. Synchronization implies that the content is being kept up-to-date between the replicas at a reasonable time-scale and logic.

Privacy There are increasing demands for privacy by Internet users. The benefits of these applications are that they only use encrypted transports and encrypt the data that is stored at third parties. This makes it more unlikely that eavesdroppers with access to transport routes or community infrastructure are able to reveal sensitive data.

3.2 Requirements

Based on the given use cases, we now can identify a set of requirements that a user-centric solution should fulfill.

Unique User Identity Every user of a community must be uniquely identifiable so that identity names can be employed to identify content.

Verifiable Identity Also, user identities must be verifiable so that others are able to verify the alleged identity of the remote peer they are communicating with.

Multi-presence It should be possible for a user to stay online in the community with several clients at the same time (multiple presence), so that content can be served from different hosts belonging to one user.

Unique Content Identifier Content must be uniquely identifiable in the sense that a document must get a unique, persistent handle to be shared between users. Such an identifier must be uniformly constructed and generic enough to support the use cases described here as well as future usages.

Location-independent Identifier The identifier must be constructed in a way that is not tied to a specific host. It should rather be host-independent to be able to shift content between hosts and decouple the content names from the underlying infrastructure as introduced by ICN. Content can then be published independently of the actual peer serving it, thus enabling flexible offloading approaches.

Seamless Handover A solution is required to take into account peers changing networks quickly. Web applications need to implement transparent and quick handovers. Since the identifiers are host-independent, the host resolution must be flexible enough to support quick updates to host locations.

Content Replication Content may be shared among peers but must be available through one identifier, which shall be resolvable to more than one host address.

3.3 Solution Concept

After laying out typical use cases on the Web and specifying the requirements for our concept, we now continue by presenting our solution concept.

3.3.1 ID Assignment

in BOPlish, every peer has two unique IDs: The peer ID is a location-dependent identifier for the peer used similar to an IP address for routing purposes. It is temporary in the sense that a re-connect of the peer (e.g., after losing network connectivity) may result in the peer being assigned a different peer ID. The peer ID has three purposes [51]:

1. To address the node itself.
2. To determine the node's position in the DHT and to route data to that destination.
3. To determine the data set for which the node is responsible.

The BOPlish ID, on the other hand, is a permanent unique identifier tied to a user's identity and not to a peer. It is location-independent and resembles a DNS name. This ID is user-friendly so that it may be shared between users without misunderstandings (such as an e-mail address or a domain name).

The assignment process for either of the two IDs needs to be secure in the sense that no user can (intentionally or unintentionally) be assigned to an ID that already refers to another user. Approaches to secure ID assignment and identity verification in structured P2P networks have been previously proposed, e.g., in RELOAD [51]. Assigning identifiers to peers in a secure way is discussed in several publications ([94] gives a survey) and bears the difficulty that no host shall be able to (intentionally or unintentionally) be assigned an ID that already refers to another host. Castro et al. describe the problem space as such: "Secure nodeId assignment ensures that an attacker cannot choose the value of nodeIds assigned to the nodes that the attacker controls. Without it, the attacker could arrange to control all replicas of a given object, or to mediate all traffic to and from a victim node." [14] Attacks, in which an adversary achieves to be assigned a multitude of IDs, are called Sybil attacks [27].

Three possibilities are most commonly suggested:

1. Peers self-assign IDs randomly (no security).
2. Ids are assigned to peers by a central authority (trust anchor).

3. An implementation of Identity-based Cryptography is employed.

While the first option provides no security at all, the remaining two alternatives let other peers verify the ID of the peer they are communicating with. BOPlish does not demand one specific mechanism. Rather, we leave the specifics of ID assignment up to the implementation, depending on the security needs of the users as well as the complexity of the software architecture. It has to be noted, though, that centralized approaches such as the usage of a trust anchor may contradict the goal of as much decentralization as possible.

3.3.2 The BOPlish URI Scheme

For the purpose of sharing and accessing content, the design of content identifiers is a key ingredient. We start from URIs, the common meta-scheme for Web resources. For the further specification, we follow three steps. First, we build on the recent Common API for (multicast) publish/subscribe [99]. RFC 7046 provides a standard syntax for an identifier of the form `id@instantiation` along with security credentials. Second, we center IDs around users that are 'instantiated' by identity providers. Third and last, we add the name of the application-layer protocol (instead of ports) to facilitate a transparent communication context.

In summary, our proposal for a uniform content naming reads:

```
bop:username@idp:protocol[/path[?parameters]]
```

These content URIs are comprised of the scheme `bop` and a hierarchical component further built from a unique `username` verified by an identity provider `idp`, followed by a `protocol` and `path` specifier and optional `parameters` that can include security credentials. The protocol specifier is used for setting different usages in one community, e.g., a chat service and a document sharing service. A peer uses that identifier to pass the URI to different modules of the application. This puts part of the application-specific semantics into the URI, with the consequence that not every BOPlish application may be able to serve every URI. The advantage of this design is that BOPlish URIs are flexible and extensible enough to easily reflect future use cases. Such a URI is generated for every published item and is shared to other users. The sharing itself is done as with HTTP URLs, e.g., via XMPP, e-mail or by publishing URIs on a website. These are some examples of BOPlish URIs:

```
bop:bob@example.org:document/picard.png?sha-256;1234abc...
bop:alice@example.com:search/Music/*tomte*
```

BOPlish URIs guarantee a location-independence by employing the username instead of a specific host identifier. The actual address of a peer responsible for a specific user is resolved via the User Community itself (using the DHT). A query for one key in the DHT may result in a list of peer addresses, reflecting the currently available content publishers.

3.3.3 Name Resolution and Data Routing

Since our URIs must be location-independent, a mechanism is needed to unbind the relation between a current location (peer ID) and the content identifier (URI). ICN features such a mechanism but operates on the network layer and therefore requires deep changes to the network infrastructure. The process of content retrieval in BOPlish, on the other hand, is comprised of three steps that involve the overlay only: Resolve the authority part of a URI to a peer ID, establish a connection to the resulting peer and then transfer the data from the source to the receiver. The BOPlish overlay provides a distributed hash table (DHT) which uses a hash of the user identifier as key and a reference to the node that holds the content (the peer ID) as value. This indirection allows the system to handle names and locations separately which we identified as a requirement for a content-centric architecture above.

One problem with DHTs is that they tend to be fragile when peers join/leave the network in a high frequency [74]. The grave reason for this is the need to re-organize the key space which requires to move the DHT content from one peer to another. Our approach stores only light-weight data structures in the DHT to prevent re-organizing from having a big impact on the system. Peers only store the identifier-location linkage (BOPlish ID => peer ID), not the content itself. As a result, the DHT can be designed to be highly churn-resistant and redundant, because the transfer of key/value pairs from one peer to another requires little bandwidth.

The name resolving mechanism scales with the number of identifiers stored. Instead of spanning the Web as a whole and hold all BOPlish IDentifiers in one DHT, we define a group of users as a BOPlish User Community. Such a community consists of users with interest in specific content. E.g., if the BOPlish application is a social network, the community is defined as all users of the social network.

After the name resolution mechanism found a location for the requested URI, the data has to be routed between the communicating peers. Data routing in the BOPlish architecture is decoupled from the name resolution overlay. Instead of using the reverse path of the name resolution, BOPlish opens a direct WebRTC connection between the content receiver and one or more of the publishers. Coupling the data routing with the name resolution is also possible but routing the content through the DHT would impose unnecessary load, leading to poor performance regarding the name lookup. Moreover, depending on the DHT implementation, the

overlay path can be disadvantageous because it is not aware of geographical and performance properties of the overlay hops. If the connection to the publisher fails, the content receiver can always re-query the DHT to find the updated location information. This allows for mobility of both, the content receiver and the publisher because the DHT entry can easily be updated without requiring a name change of the content's identifier.

3.4 BOPlish High-Level Overview

After introducing our motivation, outlining use cases/requirements and describing our solution concept, we exemplify a common situation using a high-level view of our BOPlish approach. The elements of our solution concept are composed into an integrated architecture depicted in Fig. 3.1. In the example, a BOPlish User Community containing four peers has been established (with BOPlish IDs: alice@example.org, bob@example.com, ...) using the Bootstrap Server in the upper right corner.

A DHT is formed and every BOPlish ID is mapped to a Peer ID (1, 3, 6, 9) using the ID assignment process. This mapping is stored in the DHT and accessible to all peers via the Name Resolver API. Optionally, every peer can validate other peers using the identity provider denoted in the authority part of the remote peer's address (e.g., `example.org`).

Alice publishes BOPlish URIs which Bob can use to gain access (e.g., `chat/room1`) by first resolving Alice's BOPlish ID to a peer ID (alice@example.org $\mapsto$ 3). Afterwards, a Data Channel connection is established between the two peers by routing the corresponding offer/answer messages through the DHT. Bob can now communicate with Alice using the chat protocol via the protocol-specific API. This API is defined by the application building on top of BOPlish (e.g., a chat application). The remainder of the URI's path is passed to the application as a list of parameters.

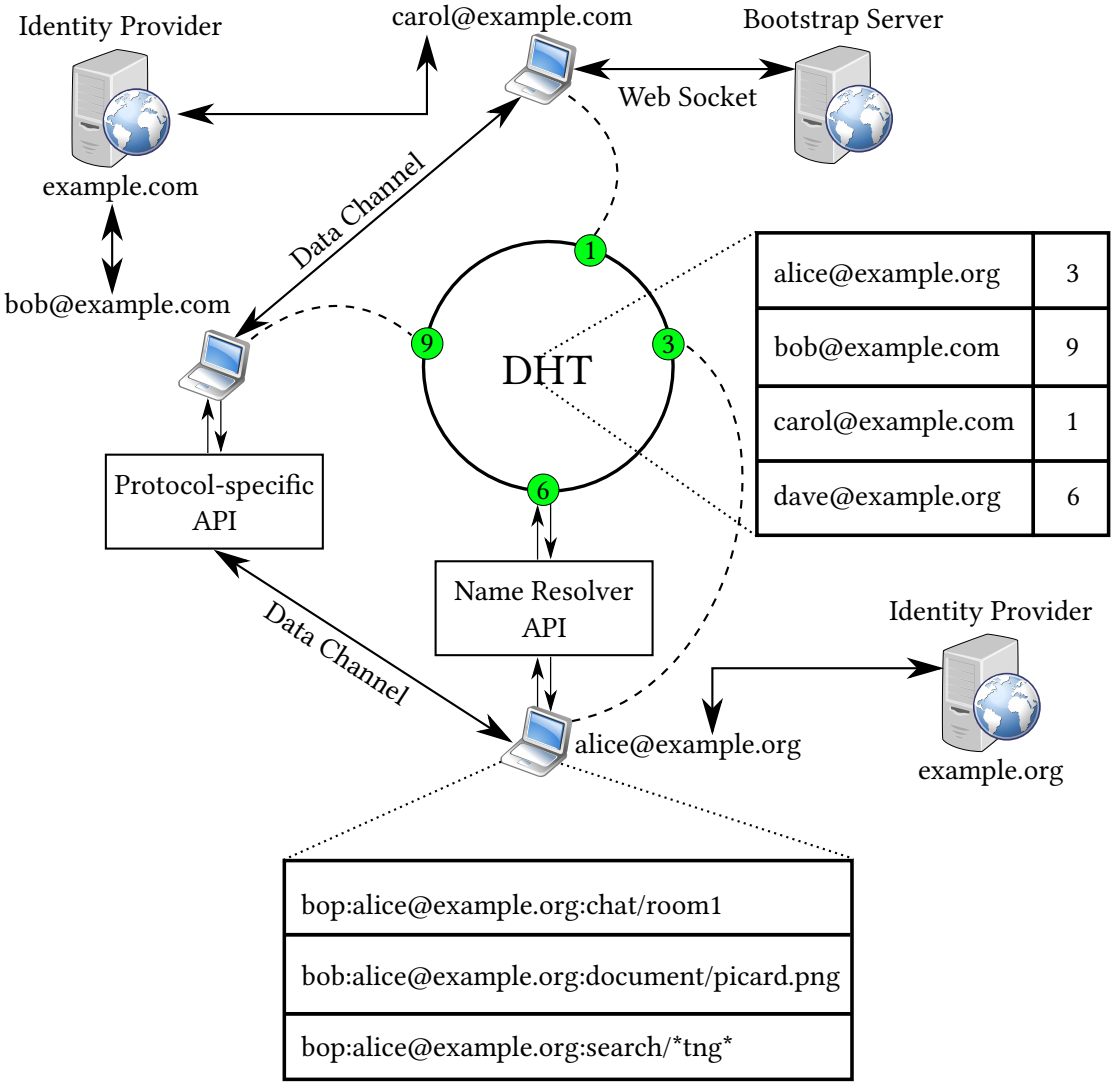


Figure 3.1: High-level view of a BOPlish User Community

4 Implementation

After laying out the concepts that make up BOPlish in Sec. 3 we now proceed to describe the reference implementation. BOPlish is supposed to be run in Web browsers. As such, our implementation is bound to JavaScript. Apart from the core library that runs on each peer, other components are needed:

- A Bootstrap Server acting as rendezvous point
- An emulation/test environment
- Demo applications that leverage the BOPlish infrastructure

The JavaScript client library (further referred to as core library) can be included in Web applications either by running directly in the browser or potentially on a server using a JavaScript runtime environment like Node.js[1]. A user navigates to a Web page and automatically joins the User Community. After the user has joined the overlay network, he can request content or publish content himself. This overlay could even span across Web sites so that a user that joined from `example.org` can communicate with a user from `example.com`, given that both bootstrap servers cooperate. This allows for a decentralized, domain independent content distribution which is not tied to central services. BOPlish uses WebRTC as its transport mechanism, allowing for direct peer-to-peer connections between the clients' browsers.

We developed our BOPlish implementation from the bottom-up, with each step allowing us to gain more insight into how WebRTC specifics hinder or enable our attempts. The first milestone enabled us to build P2P applications like a chat and a simple game [103]. We built a server component for bootstrapping each peer and connecting new peers to existing ones. On the client-side we implemented a Connection Manager for abstracting the process of establishing new WebRTC connections in a P2P network as well as a Router component that forwards messages to the correct peers by spanning a full mesh network. Then we performed the next steps in order to implement our vision of a generic user-centric, infrastructure-independent content-sharing facility. We exchanged the full mesh Router component with a Chord DHT

[1] `https://js-platform.github.io/node-webrtc/`

implementation, added the functionality to handle BOPlish URIs and established a baseline for automatic testing of our code.

4.1 Code Organization

All of the BOPlish source code is published on Github[2] under the BSD 2-clause license[3]. We have organized the code base into five different projects:

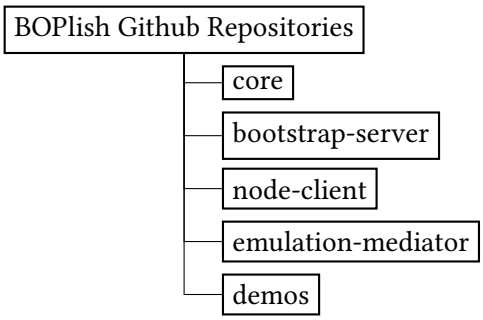


Figure 4.1: Directory tree of the main projects

The `core` project contains the code that is to be run on each peer. It is distributed as a single minified JavaScript file which can easily be included in any Web application. As browsers are the main entities in BOPlish, the core library contains the crucial components that form a User Community.

Every User Community is centered around a bootstrap server instance which can be found in the `bootstrap-server` project. We created two independent implementations, one written in JavaScript using Node.js, and the other written in Python using Flask[4]. Sec. 4.6 dives into the details of the server.

The repository `node-client` contains a module for running BOPlish peers inside of Node.js. It provides an HTTP API for creating and deleting peers as well as gathering statistics about running peers. This API is used by the fourth project, the `emulation-mediator`. It can be used to emulate large User Communities and gathers statistics about throughput, delay, bandwidth and further performance characteristics (see Sec. 4.7).

Throughout the development of BOPlish, we created a multitude of demo applications to showcase the capabilities of BOPlish. These demos are supposed to give guidance to developers that want to use BOPlish in their own projects and demonstrate the feasibility of our approach.

[2] `https://github.com/boplish`
[3] `http://opensource.org/licenses/BSD-2-Clause`
[4] `http://flask.pocoo.org/`

A selection of these applications is described in 4.8 while the source code resides in the `demos` repository.

4.2 Build Environment

JavaScript code is compiled just-in-time by the browser that executes it. Still, the source code needs to go through a series of tasks before being usable for which we used a multitude of third-party software. The most important components will be discussed now.

Source Code Management We used a shared git repository for revision control purposes. The first iterations of our code were stored on a private server and after releasing the code on GitHub, we now work on a public git 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 run tests by calling `grunt test` or to build a distribution JavaScript file by calling `grunt dist`.

Testing Strategy Since the very beginning of the BOPlish project, we created unit-tests alongside 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.

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.

[5] `http://gruntjs.com/`
[6] `http://visionmedia.github.io/mocha/`
[7] `http://sinonjs.org/`
[8] `http://usejsdoc.org/`

4.3 Software Architecture

The design of our architecture is presented in Fig. 4.2. At the very top sits the BOPlish application which provides a simple interface for developers building their applications on top of BOPlish. The developer facing part uses the BOPlish Core API to send and receive data and controls the bootstrap process. Moreover, it instantiates a Router and a Connection Manager which handles WebRTC-specific connection establishment and management.

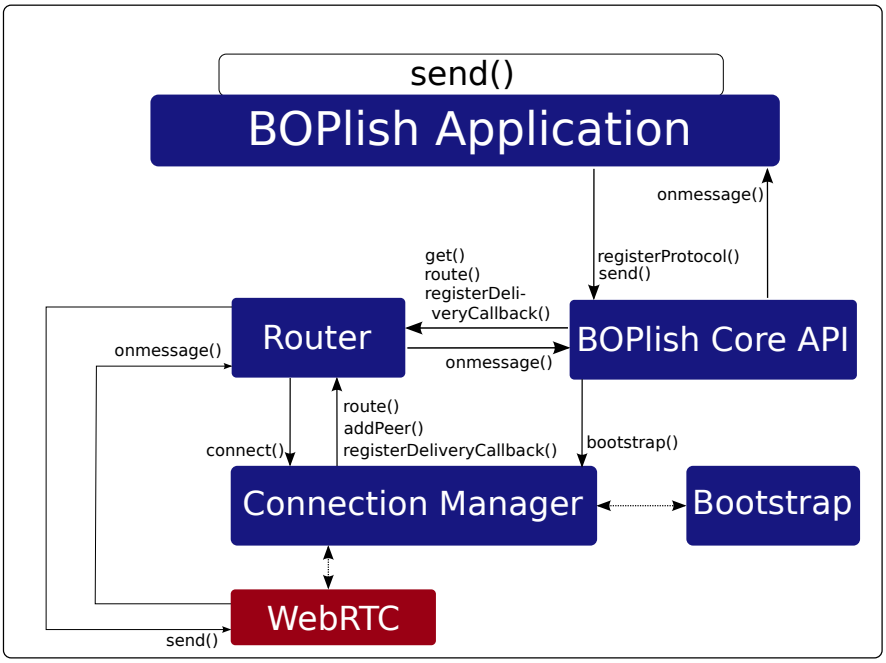


Figure 4.2: Overview of the BOPlish software architecture

The Router component is responsible for deciding where to forward messages to and thus maintains a routing table. It exposes a KBR API as defined in [22] that hides the DHT implementation introduced in Sec. 4.5. The API allows us to easily exchange the underlying P2P protocol. Our first approach included a full mesh which is still usable for small communities [103]. The Router encapsulates messages into the routing format (see Sec. 4.3.3) and maintains the connection to the bootstrap server to recover in case of failures and during bootstrap.

The Connection Manager component is responsible for handling WebRTC specifics like connection establishment and maintenance. Contrary to our previous work, it does not maintain a list of open connections alongside the Routers peer table. Instead, the Router is the only component keeping track of open connections thus reducing complexity. To be able to join a P2P network, a node has to know at least one other node already part of that network.

The Bootstrap component encapsulates the functionality for discovering an initial node to connect to. Since this process is tightly bound to the generic connection establishment in our WebRTC-based implementation, we included this component into the Connection Manager.

4.3.1 Related Work

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 [22]. 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 promising. Hints on how software applications leveraging this approach are to be structured, however, are out of scope of the paper. Especially in the context of WebRTC, connection establishment and maintenance require a great deal of effort because of the offer/answer mechanism of JSEP. The paper does not give recommendations of how the KBR implementation shall interact with the underlying network, be it IP or – as in our case – WebRTC Data Channels. Still, the BOPlish approach complies to the API proposed by Dabek et al.

A more implementation-specific paper is OverArch [6]. The authors propose a detailed architecture for structured and unstructured overlay networks. Building on top of Dabek's work of 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 makes it easy 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). Moreover, an application could choose from a specific routing implementation. OverArch specifies even the inner workings of the KBR module which helped us putting our implementation to work.

4.3.2 Data Transport Topology

The transport of data in BOPlish is comprised of two layers: an upper generic layer used for passing application-specific messages and a lower routing layer specific to the Router implementation. The messages on the routing layer are exchanged either via WebSockets through the bootstrap server or via WebRTC Data Channels in a hop-by-hop manner. The messages on the application-specific layer transparently facilitate the routing layer to reach the receiving peer (possibly passing a number of other peers used as intermediate hops).

Due to the nature of WebRTC, a permanent rendezvous instance is needed that maintains connections to at least one active peer. That instance must be constantly reachable (online) and uniquely addressable (e.g., via a known URI/URL). We call this instance the bootstrap server. It is used to transfer the initial signaling messages from a newly joined peer to a chosen existing peer. Since BOPlish applications are served from a Web server we chose to use a Web server together with the WebSocket protocol as bootstrap server. Furthermore, on the routing layer that manages the overlay, a bootstrap node has to be chosen to initialize the joining procedure. Thus, from here on we differentiate between a “bootstrap server” and a “bootstrap peer”, where the former is used to open a WebRTC data channel to the latter.

4.3.3 Message Format

As described above, BOPlish consists of two layers. The Router communicates with other Router components of remote peers and the bootstrap server. The format of routing layer messages looks as follows:

```
1 {
2  to: "<Peer ID receiver>",
3  from: "<Peer ID sender>",
4  type: "ROUTE",
```

```
 seq: "<seq nr>",
 payload: {
    <JSON-formatted payload>
 }
}
```

Listing 4.1: BOPlish JSON-encoded message format on the routing layer

Such a message consists of `to` and `from` fields which denote the sender's and receiver's Peer ID. Moreover, a message that shall be routed to other peers has its `type` field set to ROUTE. Depending on the router implementation, other semantics for this field may be defined. The message also carries a sequence number used as a transactional ID because of the asynchronous nature of the communication channel. At last, the message contains a payload set by the upper layer.

Messages on the upper layer can be divided into two main categories: messages that are used for WebRTC signaling (we call this the `signaling-protocol`) and other messages send by the application-layer protocols defined on top of BOPlish. The `signaling-layer` messages are further defined in Sec. 4.3.4. Generic messages passed through the BOPlish API look like this:

```
{
 to: "<bop ID receiver>",
 from: "<bop ID sender>",
 type: "<protocol-identifier>",
 payload: {
    <JSON-formatted payload>
 }
}
```

Instead of denoting the Peer IDs, application-layer messages contain BOPlish IDs and a `type` field that maps the message to the appropriate protocol (as described in 4.4.1). Again, every protocol can set a custom payload using the `payload` field.

4.3.4 Bootstrap Procedure

Every peer in a BOPlish user network first acquires a unique ID (the Peer ID) using one of the mechanisms stated in Sec. 3.3.1 and establishes a WebSocket connection to the chosen bootstrap server using that id. Afterwards, a new peer sends a bootstrap message to the bootstrap server (Fig. 4.3 shows the complete process). The generation of that message is triggered by the API call `ConnectionManager.bootstrap()` which gets called automatically once the

WebSocket transits to an open state. In the current state, the peer does not know any other peer in the system. Thus, the bootstrap message can not contain a recipient. Such messages are denoted by the "to" field set to the value "*" and the "type" field set to "signaling-protocol". This message instructs the bootstrap server to choose the bootstrap peer. The payload contains the offer generated by the WebRTC API call `PeerConnection.createOffer()` (which is a standardized RFC-3264 offer [77]). The first message sent by a new peer thus looks like this:

```
{
  to: "*",
  from: "<Peer ID sender>",
  type: "signaling-protocol",
  payload: {
    type: "offer",
    offer: "<offer SDP>"
  }
}
```

Listing 4.2: BOPlish bootstrap message

If there is no other peer connected to the bootstrap server, the server answers with a message where the "type" field is set to "denied". In this case the peer has to wait for an initial connection establishment by a second peer (indicated by an incoming offer; this procedure is described in detail in our PJ1 report [103]). When joining an existing network (with at least one peer), the initial offer is routed through the bootstrap server to one of the peers. The algorithm employed to choose which peer receives the initial offer is up to the server and may range from pure random selection to more refined algorithms which take into account the online time or authorization credentials. We currently use a plain random approach.

The chosen peer, after receiving the initial offer, answers with a signaling message of type "answer" and the corresponding answer SDP [77]. This answer is routed through the bootstrap server and the two peers establish a Data Channel connection. From then on, the Data Channel is the only transport channel used by the new peer for exchanging messages with other peers. The bootstrap server (and thus the WebSocket connection) does not have an active role anymore (besides connecting newly joined peers to existing ones). The new Data Channel object is passed from the Connection Manager to the Router component. The Router now uses that Data Channel to initiate the bootstrapping of the routing protocol, in our case Chord.

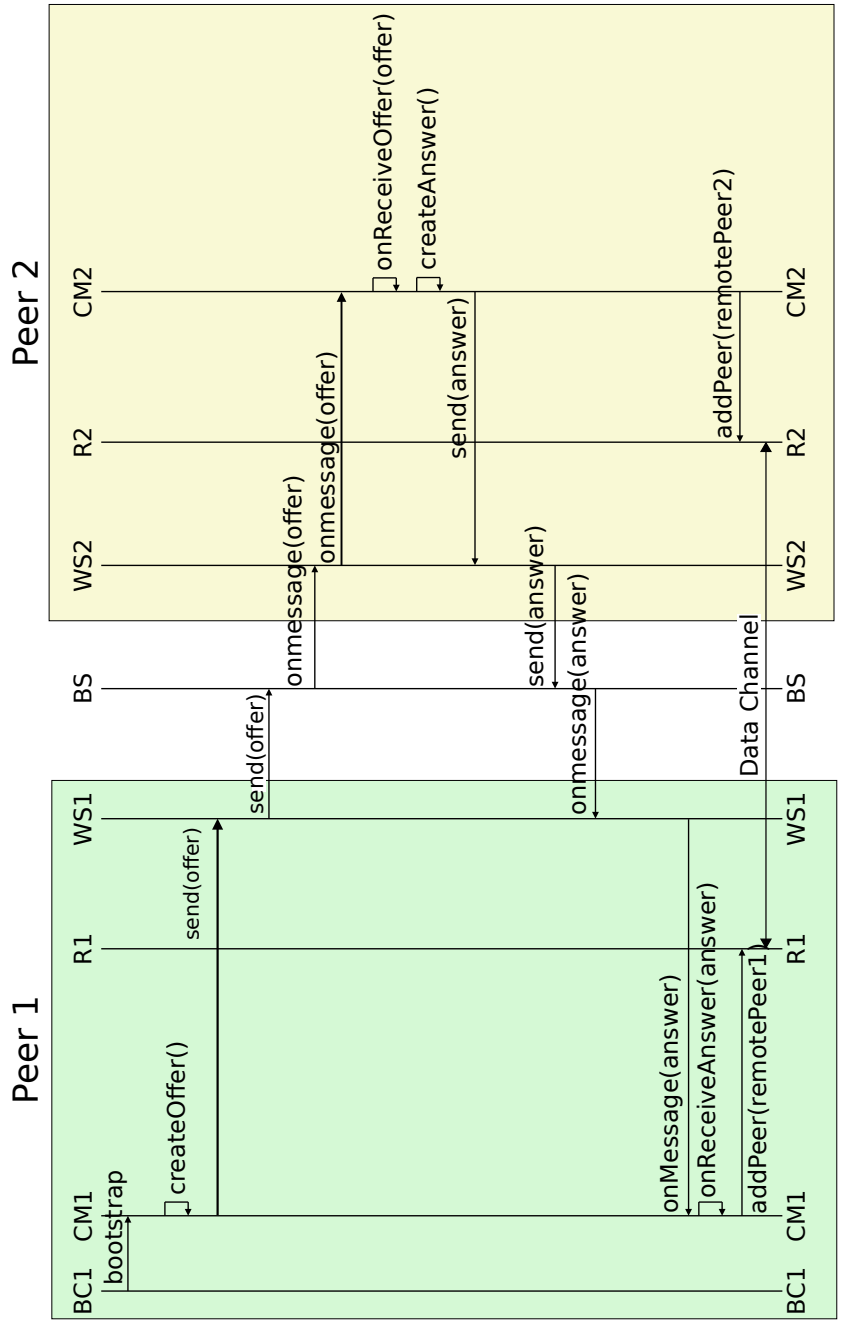


Figure 4.3: Sequence diagram of the bootstrap process in BOPlish. Peer 1 generates an offer, sends it through a WebSocket connection to the bootstrap server which then selects a candidate used for bootstrapping, in this case, peer 2. Then, peer 2 generates an answer, sends it back through the bootstrap server, eventually resulting in a Data Channel between the two peers.

4.4 Main Building Blocks

After describing the general software architecture, we can dive deeper into the specifics of the BOPlish implementation. To do so, we start by giving an introduction to the application developer-facing API before diving into each component more thoroughly. These components include the Peer, Connection Manager and Router classes.

4.4.1 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, the skeleton of 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 one main function, namely `registerProtocol()`.

A client protocol has to have a distinct name (in the scope of the code base) with which it is identified in the `Routing`-Component. In our example, the protocol is called ping-protocol. The application developer registers the ping-protocol in the BOPlish core by calling the factory-method `registerProtocol()` with the protocol name as its argument as shown below. This call returns a protocol object with predefined functionality for sending and receiving messages. This ensures that the different developer-defined protocols cannot accidentally send messages directed to other protocols.

```
1 var bopclient = new BOPlishclient('wss://example.org/');
2 var pingProto = bopclient.registerProtocol('ping-protocol');
3
4 pingProto.onmessage = function(msg) {
5   if (msg.type === 'ping') {
6     pingProto.send(msg.from, {
7       type: 'pong',
8       date: msg.date
9     });
10  } else if (msg.type === 'pong') {
11    console.log('ping took', new Date() - msg.date, 'ms');
12  }
13 };
```

Listing 4.3: Defining an application-layer protocol in BOPlish

This code registers the ping-protocol in the `Router` which can then identify incoming messages by the protocol name and call the corresponding onmessage-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 `onmessageHandler` is called by the `Router` and (in this case) answers with a `pong`-message by calling the `send()`-method on the `protocol`-instance with the recipient ID and the payload. The ping-protocol is started by sending a `ping`-message to the remote peer that is to be pinged:

```
1 pingProto.send('bop@id.org', {
2   type: 'ping',
3   date: new Date()
4 });
```

The `ping-protocol` described above is a simplified version of the implementation found in the demo repository (see *protocols.ping.js*). Other protocols we implemented that are used by either the demo applications described in 4.8 or the core library itself are 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 (4.4.4) using the identifier `topo-protocol`. It defines the subtypes `request` and `response`.

Bopcast Protocol

The bopcast protocol is a communication protocol that handles group management and creates a full mesh data distribution graph. Using this protocol, a simple application-layer multicast can be implemented. Fig. 4.4 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.

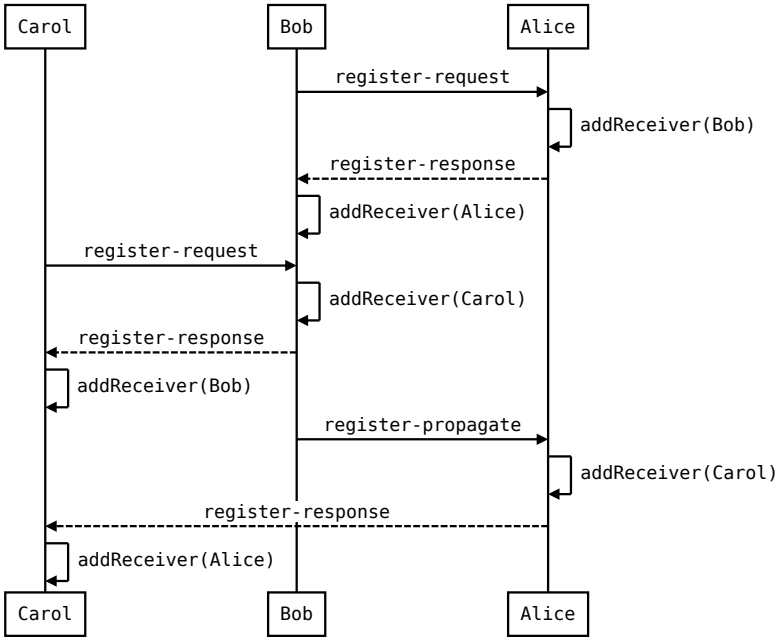


Figure 4.4: Bopcast sequence diagram showing the process of forming a group of peers

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 (4.4.4) using the identifier `bopcast-protocol`. It carries the subtypes `register-request`, `register-propagate`, `register-response` and `deliver`.

4.4.2 Peer

A `Peer` models a remote BOPlish node in the system. Every peer holds a list of other `Peers` that have open Data Channel connections between each other. Once the Data Channel closes, the `Peer`-instance is deleted. Such `Peers` are instantiated by a `Connection Manager` and stored in the routing table of the `Router`. To keep the implementation simple, the `Peer` is kept as slim as possible. Along with the WebRTC channel references it contains an `id` that

identifies the remote peer and `send` as well as `onmessage` methods.

Along with this basic wrapper functionality the `Peer`-class employs an heartbeat mechanism to check whether the connection to the remote peer is still alive. This is necessary as we found the current Data Channel implementation to be unable to show the correct state of the connection. As an example, it is not possible for a Chrome browser to detect whether the connection to a Firefox browser has been closed or not using the WebRTC API even though this is specified in the specification[9]. The heartbeat mechanism is configurable in both, the timeout threshold before the connection is marked unavailable and the interval in which heartbeat messages are sent. We plan on disabling this functionality as soon as the browser implementations mature, further reducing the introduced overhead.

4.4.3 Connection Manager

The Connection Manager (residing in `connectionmanager.js`) is responsible for establishing WebRTC Data Channels between peers. Every peer holds a single 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 Sec. 4.4.4 for details of how messages are forwarded). Upon receiving an appropriate answer (again through the router), 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).

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). Since all connection establishment in WebRTC is asynchronous, the Connection Manager must store the state of every connection and act appropriately on situations like connection loss, no answer received, glare (simultaneous offer from two peers).

4.4.4 Router

The Router component constitutes the heart of the core BOPlish library. It is responsible for disseminating messages through the network. Additionally, it maintains the connection to the bootstrap server and maintains a routing table consisting of `Peer` instances. These methods make up the public API of the Router:

- `addPeer(peer)`

[9] http://www.w3.org/TR/webrtc/#widl-RTCDataChannel-onclose

- `removePeer(peer)`
- `route(to, message, callback)`
- `forward(to, message, callback)`
- `put(key, value, callback)`
- `get(key, callback)`
- `registerDeliveryCallback(protocol-type, callback)`

Since the other components don't make any further assumptions about the router it can be exchanged quickly and easily. Our current implementation is capable of creating a fully meshed network as well as an DHT implementation depending on the used router implementation. The DHT implementation is described more thoroughly in Sec. 4.5.

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:

```
{
  to: "f9c89ceb726436bb0d7074c08788d08e0e974dbf",
  from: "48142a86bd1dc7c2be81df1c5ca6d0a98c328f4b",
  type: "ping-protocol",
  payload: {
    "date": "Mon␣Nov␣11␣2013␣17:44:05␣GMT+0100␣(CET)",
    "type": "pong"
  }
}
```

Listing 4.4: Example for a ping-protocol message

4.5 DHT Implementation

As described in Sec. 3, we use a Distributed Hash Table (DHT) as a name resolving mechanism. In this section, we provide a detailed look into our JavaScript DHT implementation. We opted for the Chord protocol [90] since it is straight forward to implement and the paper provides detailed implementation information and pseudo-code. Also, Chord has been selected for RELOAD [51], which suggests its qualification even in large-scale deployments. Exchanging

Chord with another protocol is unproblematic due to the standardized KBR API that BOPlish uses. In the future, developers may wish to implement, e.g., Kademlia, which is known for its implementation-friendliness, too [59].

BOPlish differs from typical implementations due to the underlying browser environment. All existing Chord implementations we know about (such as the widely known OpenChord[10]) use a socket-based API (based on TCP or UDP) or Remote Procedure Calls (RPC). Peers are addressed directly, provided the target IP/port combination is known. BOPlish uses WebRTC, which does not allow such a direct connection establishment. Instead, signaling information has to be exchanged using an external channel prior to any data transmission. Standard WebRTC use cases are intended to use a centralized signaling server for that task. BOPlish applications shall not rely on such central infrastructure. Therefore, instead of reintroducing centralized components, we shifted the signaling functionality to the DHT layer, using that layer for connection establishment itself. Other changes compared to the original Chord implementation were made to adapt to the environment:

- BOPlish uses recursive instead of iterative routing due to the cost of connection establishments (i.e., the exchange of signaling messages).
- The Chord finger table (comparable to a routing table) consists of 160 entries (given, SHA-1 is used), each containing a peer identifier (a hash) and an IP/port combination. BOPlish uses a dynamically sized finger table (to minimize the number of open connections, which is a constraint imposed by current browsers) and stores peer IDs instead of IP/port combinations.

As a future addition to BOPlish, we plan to extend the single predecessor and successor entries proposed in the original paper to include a list of entries. In the remainder of this chapter we provide a detailed look into the BOPlish Chord implementation while focusing on the changes made to the original Chord protocol.

4.5.1 Software Overview

As pictured in Figure 4.5, every Chord node that is known to a local Chord instance is represented by a `Node` object. When a `Chord` instance is created, it instantiates a `Node` and stores it as `local_node`. This local node object stores the direct successor and predecessor of the Chord instance and is used to find its entry point in the Chord ring. Another specialty of the

[10] `http://www.uni-bamberg.de/en/fakultaeten/wiai/faecher/informatik/lspi/bereich/research/software_projects/openchord/`

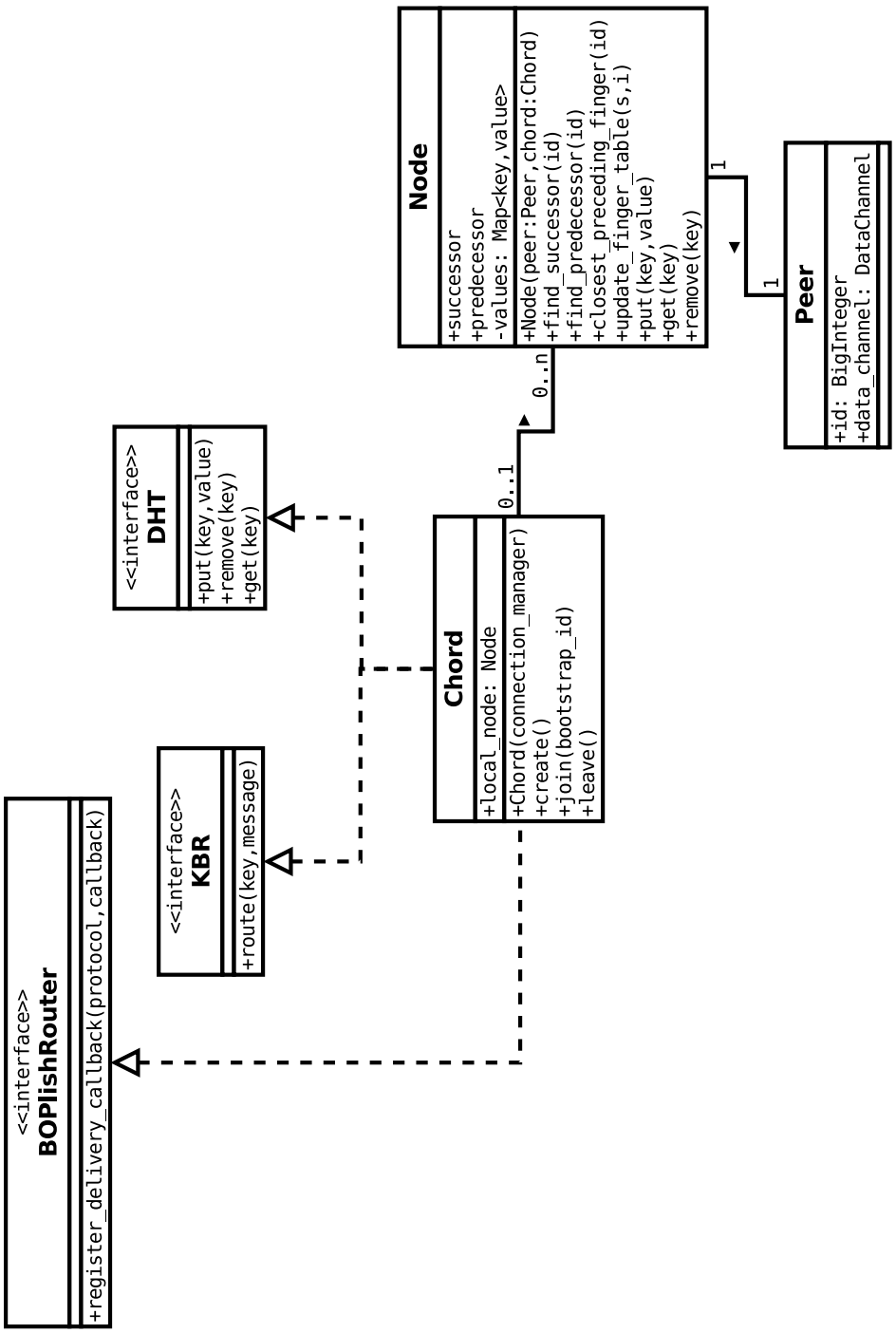


Figure 4.5: Class diagram of the BOPlish Chord implementation

local node object is that it does not carry a WebRTC Data Channel but the WebSocket channel to the bootstrap server. This has two purposes:

1. For joining an existing Chord ring, the bootstrap server is the only known rendezvous instance.

2. New nodes joining the ring send their initial offer through the bootstrap server, using this Chord instance's local node as end point.

The fact that the local node carries a WebSocket instead of a Data Channel does not have any effect on the generic implementation of our node class because both expose the same interface (i.e., `onmessage` and `send()`).

For every node in the finger table, such a `Node` instance is created, encapsulating the transport channel (a WebRTC Data Channel) and exposing RPC-style methods for finding its successor and predecessor, updating the finger table etc. In our implementation, nodes put together the specific JSON message for every method call and send this message to the remote node which then handles the specific request (e.g., answering with its successor). This way we have achieved an easy-to-use abstraction between transport messaging and Chord logic, which makes it easy to implement the various Chord semantics.

Due to the nature of the asynchronous WebRTC Data Channel interface, we had to cope with the fact that responses to requests sent to a remote node can arrive at any point in time. This leads to a situation where multiple requests are waiting for a response. One solution to this would be to always have only one outstanding request waiting for a reply. However, this introduces the problem commonly known as head-of-line block: One long-lasting request may block the whole communication process, blocking all other requests in the queue. Thus, we implemented a way to map responses that arrive through the Data Channel to the requests sent earlier. We introduced a mechanism that uses sequence numbers for marking transactions (consisting of a request and a response). This technique is similar to SIP transactions [78]. Every Chord message carries a `seqnr` field and the corresponding request callback is saved in a local map. When the remote node eventually sends its response carrying that same sequence number, the callback is retrieved and removed from the map and then called. This makes it easy to issue multiple requests in a row without blocking the application flow.

It is possible that other Chord nodes choose the current local Chord instance as bootstrap or otherwise would like to establish a connection (e.g., for issuing a "PUT" or "GET"). In this case, such node instances are stored in a map carrying the remote node's ID as key and the node object as value. This remote reference list is checked frequently and cleaned, so that it does

not grow to an unbearable size over time. This is especially important because the number of open WebRTC Peer Connections is constrained by the browser.

The decision between recursive and iterative routing (as explained in Sec. 2.1.2) fell for recursive routing for the following reasons: In an iterative routing scenario, a Chord instance would have to open a transport channel to every peer on the path to the target peer. In our scenario, where we use WebRTC Data Channels, opening connections to another peer is a very costly operation due to the offer/answer model (as, e.g., opposed to IP. See Sec. 5 for a quantitative analysis). Thus, we implemented recursive routing so that a peer asks one of the known peers in its finger table (to which it ideally has already an open Data Channel) to route the message. The next peer then uses its open Data Channel to the next hop for further routing and so on. The answers are then passed on the exact reverse path through the already open Data Channels. In this way, we minimized the overhead for opening new transport channels. This, though, comes at the cost of having to maintain timeouts, e.g., when an intermediate peer disappears.

4.5.2 Application Programming Interface

As shown in the class diagram of Figure 4.5, our Chord implementation provides four distinct interfaces: First, there are the Chord-specific API calls for creating, joining and leaving a Chord network. Second and third, a Key-based Routing (KBR) interface as well as a Distributed Hash Table Interface for storing and querying for key/value pairs are provided; both APIs are compatible to the API proposed by Dabek et al. in [22]. The fourth interface is specific to our usage in BOPlish and is called the `BOPlishRouter` which exposes the method `register_delivery_callback()`. The usage of this method is explained in [103], where it is called `setOnMessageHandler()`. A sample usage of the DHT API is illustrated in the following code snippet:

```
1  var chord = new Chord();
2  chord.join(42, function(err) {
3     if(err) {
4        throw new Error("Error joining");
5     }
6     chord.put(12345, {
7           name: "Hans Blix",
8           profession: "politician"
9        },
10       function(err) {
11          chord.get(54321, function(obj) {
```

```
12              console.log(obj);
13          });
14    });
15 });
```

Here, a Chord instance is created and then this instance is added to an existing Chord network using the Peer with ID 42. Since we built all our interfaces in an asynchronous way, the `join()` call is passed a callback function which is called after the joining has (successfully or unsuccessfully) ended.

When the peer has joined, a simple JSON object is stored in the DHT with the key "12345" using the asynchronous DHT call `put()`. When this call has succeeded, the value is retrieved from the DHT using `get()` and then printed on the console.

4.5.3 Chord bootstrapping

For joining an existing Chord network, three operations have to be undertaken:

1. Initialize new node's successor with the help of the bootstrap node
2. Start stabilization interval, keeping new node's successor up-to-date
3. Start finger table fixing interval, filling finger table

First, the new peer sends the following message to the bootstrap node to retrieve its own successor:

```
1 {
2  type: "FIND_SUCCESSOR",
3  id:"<peer_id_sender>",
4  seqnr: "<sequence_number>",
5  from:"<peer_id_sender>"
6 }
```

The type "FIND_SUCCESSOR" denotes that the message is to be handled by the Chord instance (and not, e.g., by the Connection Manager). The field "seqnr" must be present on all Chord messages. Its purpose is for every Chord instance to be able to map responses (e.g., a successor ID) to requests (e.g., "FIND_SUCCESSOR") as described in Sec. 4.5.1. The "id" field contains the ID of the first finger table entry's start (which, when bootstrapping is the sender's peer ID).

The bootstrap node eventually answers with a message of the following form, providing the requester with the successor's node id:

```
{
 type: "SUCCESSOR",
 successor: "<successor_id>",
 seqnr: "<sequence_number>",
 from: "<peer_id_receiver>"
}
```

In the stabilization phase of the new node, these messages are exchanged until the finger table of the new node is completely updated. For every node in the finger table, a Chord instance now carries the node's ID as well as an open Data Channel. For a detailed explanation of the Chord joining procedure see [90].

4.6 Bootstrap Server

The Bootstrap Server has two distinct tasks: Deliver the application and act as a 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 a 'to' field of value '*' 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.

4.6.1 Python

The Python implementation using the Flask Web micro framework[11] is very simple and consists of roughly 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 `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.

[11] `http://flask.pocoo.org`

4.6.2 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 (Sec. 4.2).

4.7 Emulation Environment

We identified emulation support as a crucial requirement to measure system performance characteristics and run integration tests with large numbers of participants. A headless runtime has been created that is able to execute the BOPlish core components without the need of a browser instance. The runtime allows mixing emulated (command-line) with actual (browser-based) peers. It was initially built for unit-testing purposes but also serves as a basic building block for the emulation component.

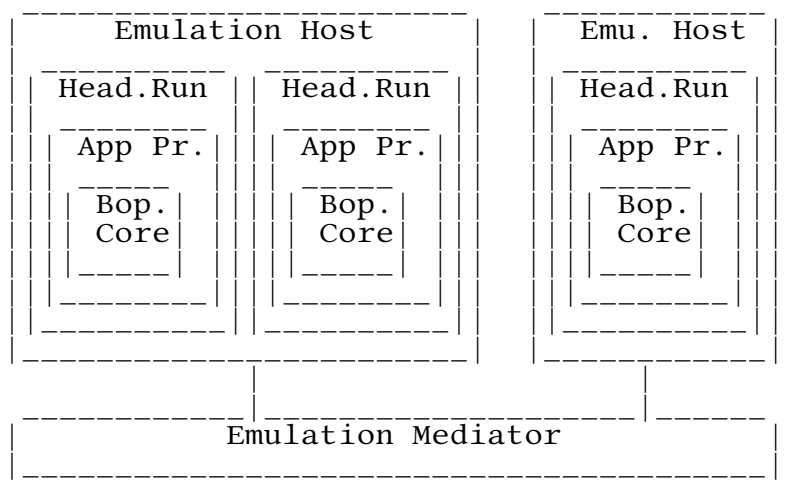


Figure 4.6: Emulation Environment Architecture

The actual emulation environment can be separated into two components: the *emulation host* is a wrapper around the headless runtime environment and the *emulation mediator*. The latter is a component that connects to the emulation hosts and centralizes logging and control.

Fig. 4.6 shows the system architecture.

Hosts participating in the emulation start an instance of the emulation host. The host can then spawn multiple workers each representing a BOPlish peer. The host connects to a mediator and sends status information from its running peers. The mediator instance communicates with all participating hosts and condenses the transmitted information in a central location. The mediator supervises the logs and notifies the user in case of errors. The user can interact with the mediator node using a administrative Web site to control the emulation hosts (e.g, spawning more workers).

4.7.1 Headless Runtime

During the evolution and growth of the project's code base it became increasingly important to test the code in an automated way. The browser platform being the main development target does not provide for a typical unit-testing environment. Code is compiled just-in-time (JIT) by the browser leading to a tedious debug process with errors occurring during runtime. To counteract these issues, a headless runtime has been developed to enable traditional unit-testing from the command-line and provide the necessary foundation for the emulation component.

Different solutions to executing code supposed to run in browsers exist. These approaches can be broadly separated into two categories: *browser automation frameworks* and *headless JavaScript runtimes*. Solutions like Selenium[12] fall into the first category of a browser automation framework. The idea is to use existing browser environments like the Mozilla Firefox or Google Chrome browser and interact with them using a pre-recorded script. The script can easily be conducted by using a browser-plugin. Approaches like Selenium work best for interaction-intensive applications that are supposed to be controlled manually by a human being. The main focus lies in the analysis of workflows from the users perspective. Even though it is possible to directly invoke JavaScript code, unit-testing abilities are limited. Moreover, using such a browser automation framework as the basis for an emulation component is problematic. Running multiple instances introduces a lot of overhead due to the entire browser environment being started for every test.

Headless JavaScript runtimes, on the other hand, aim for a different goal. Instead of relying on a stand-alone browser application, the runtime driving the browser is extracted to run on its own. The Node.js platform is an example of such a headless runtime. It is able to compile and execute JavaScript using the command line interface. On top of Node.js, test frameworks can be used to simulate interaction with the application in a generic way by using code to describe

[12] `http://seleniumhq.org/`

test cases. Mocha[13] is a widely adopted example of such a framework. Tests are written in JavaScript code and executed by using a Node.js-based test runner. This enables running tests written in code from the command line and seamless integration into JavaScript-heavy development environments.

A crucial requirement for running the same code base in a browser and in a headless environment is the support of both environments for the included third-party components. As an example, Node.js does not natively include the WebRTC technology. Instead, it can be added as a third-party addon[14]. Code changes were needed throughout the code to support cross-platform awareness. These included the introduction of a compatible module dependency system, the integration into the development environment as well as diverse code changes to rule out variation between the third-party modules on different platforms.

4.7.2 Emulation Host

The emulation host can spawn instances of the headless runtime and provides a REST API for external supervision. Messages directed to the running BOPlish hosts can be proxied to a emulation mediator via a WebSocket channel. To start an emulation host, a listen port and a BOPlish bootstrap instance have to be specified:

```
$ ./boplish-emulation-host.js --port 9000 --bootstrap\
    ws://chris.ac:5000
```

The host will then allow connections on the specified port, e.g., from a mediator instance. The REST API used to control the host is designed as shown in Tab. 4.1.

#	Method	Path	Comment
1	POST	`/peer`	Start new peer; returns `{id}`
2	DELETE	`/peer/{id}`	Shutdown/abort peer by ID
3	GET	`/peer/{id}`	Returns peer information
4	GET	`/peers`	Returns a list of all peer IDs
5	DELETE	`/killAll`	Shutdown/abort all peers
6	GET	`/status`	Return logging handler

Table 4.1: Emulation Host REST API

Peers can be started and stopped by using the API calls 1 and 2. When starting a peer, the call returns the `id` assigned to this instance. Call 3 returns information using the assigned

[13] `http://visionmedia.github.io/mocha/`
[14] `https://github.com/js-platform/node-webrtc`

`id`, while call 4 returns a list of all the peers currently running on this host. The returned `id`s can then be used to stop the peers or gather status information. It is also possible to abort all running peers at once by issuing call 5. Call 6 is a special call that upgrades the HTTP connection to a bi-directional WebSocket channel. After the connection has been established, all messages returned from the underlying BOPlish peers are sent through that channel. A Mediator, as described in Sec. 4.7.3, can therefore gather all messages in a central place. During the WebSocket initialization, a filter can be specified to prevent overloading from happening.

4.7.3 Emulation Mediator

The mediator acts as a central component in an emulation test run. It condenses the log information and supervises all participating emulated peers. The mediator consists of two parts: a backend and a frontend. The backend establishes and maintains connections to BOPlish hosts and uses a NoSQL database to store messages received from them. The mediator itself also exposes a REST-API. Interaction with that API can be automated by using a scripting language and a suitable tool like curl[15] or the frontend interface described below.

The frontend interface shown in Fig. 4.7 has been developed to simplify interaction with the mediator. The built-in chart engine can be used for custom plot generation from the gathered data. It consists of three pages: *host overview*, *host detail* and *peer detail* which will now be further elaborated on. Buttons allow a user to register emulation hosts by entering the corresponding IP and port as described in Sec. 4.7.2.

When a host is registered at the mediator, it will be shown in the host navigation menu along with the running peers on that host. More details can be gathered by clicking on the host address respectively the peer ID as described below. All the gathered data can be downloaded in JSON-encoded format for later analysis.

Host Overview

The host overview page (Fig. 4.7) gives a summary of the current overall emulation status. All the gathered data is available to the chart engine[16] which can render a multitude of different chart types and tables. Depending on the required evaluation data, the charts are supposed to quickly display information which can later be analyzed thoroughly using the gathered log files. The data available to the host overview page include:

- Current bandwidth/sec of all running hosts

[15] `http://curl.haxx.se/`
[16] based on Google Charts (`https://developers.google.com/chart`)

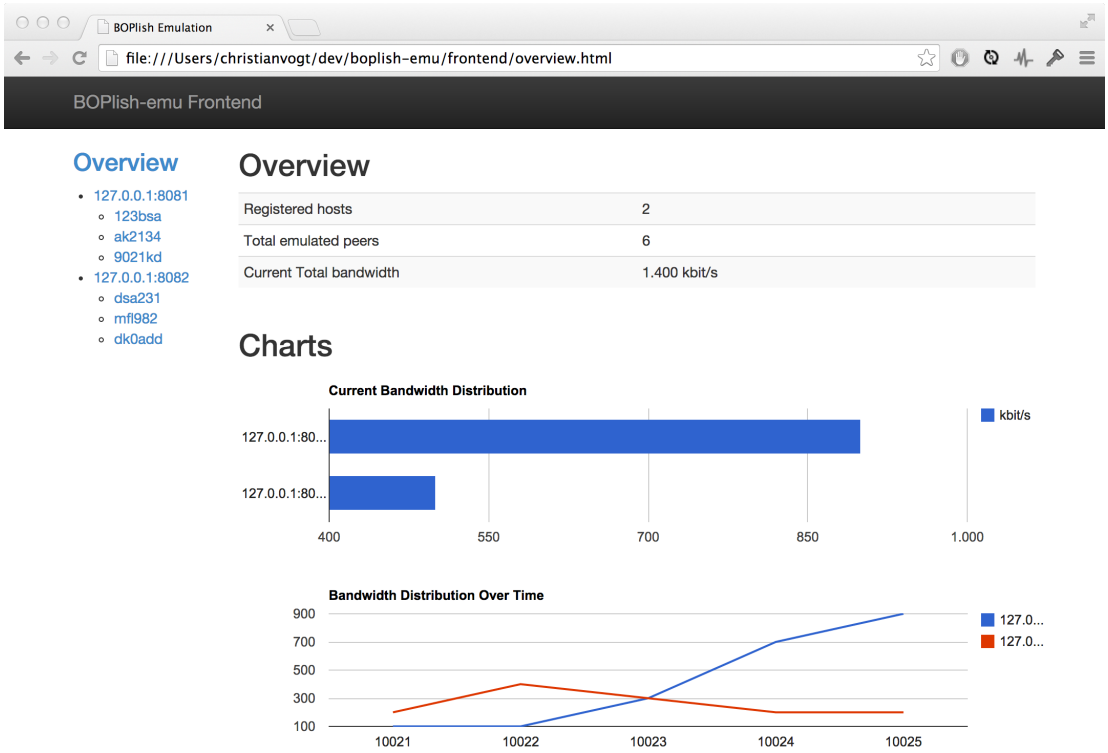


Figure 4.7: Emulation Overview Page

- Number of running hosts
- Debug info/warning/error of running hosts
- Uptime of all running hosts

Host Detail

The host detail page is revealed when clicked on a host from the navigation menu as shown in Fig. 4.8. Just as with the host detail page, the charting engine can be used to display any information available to the mediator instance. The data available to the host detail page include:

- Current bandwidth/sec of running host
- Number of running peers
- Debug info/warning/error of running peers
- Total number of received BOPlish messages

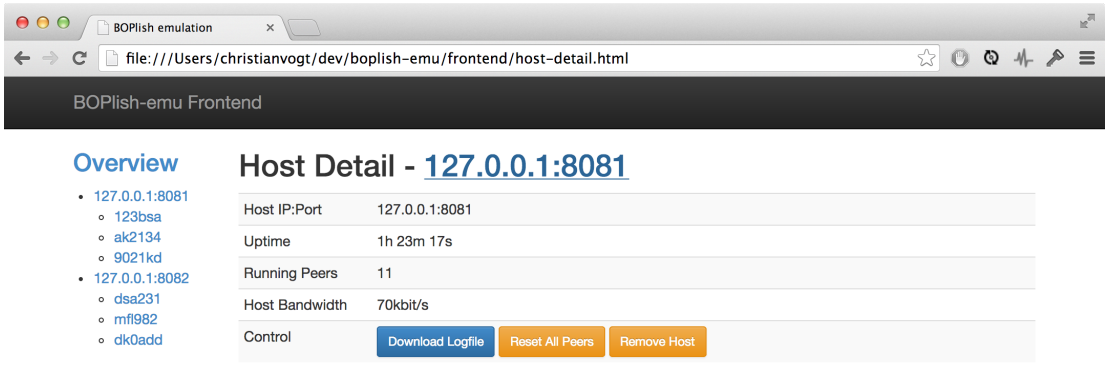


Figure 4.8: Emulation Host Detail Page

- Host uptime

Peer Detail

Clicking on a peer ID in the menu opens the peer detail page. It is supposed to show in-depth information about the running peer and help debugging if a peer fails (using the log files that still persist on the mediator). The data available to the host detail page include:

- Peer ID
- Peer uptime
- Current bandwidth/sec of running peer
- All received BOPlish messages

As stated above, all BOPlish messages occurring at every peer are stored in a NoSQL database. The database can be queried for later analysis. Every message contains a timestamp written by the host the peer is running on. It is therefore crucial to keep the time in sync among the participating hosts to keep the messages in their absolute order. For that task, NTP can be used to reduce the time difference to a acceptable level (lower than the hop-by-hop delay).

4.7.4 Issues in WebRTC Emulation

During our work with the BOPlish emulation component, we initially tried to build a generic WebRTC emulation environment that could also be used with other projects. Such an emulation component could either rely on the availability of a server-based WebRTC library or implement its own WebRTC stack. The later is considered hard, the user-space WebRTC stack contains

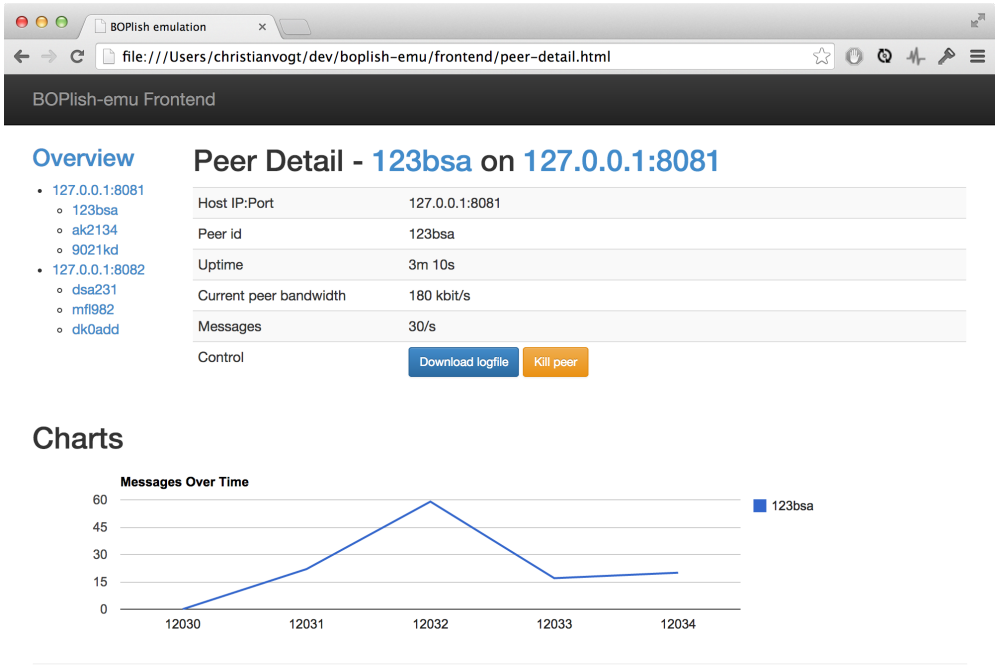


Figure 4.9: Emulation Peer Detail Page

SCTP, ICE, SRTP just to name a few. The other possibility is to rely on a third-party library. This is what we have done with our approach to the BOPlish emulation component. Unfortunately, the library revealed limitations. Bugs in the Data Channel implementation might lead to abrupt dropouts without any error messages and general system resource utilization was very high. As an example, it was only possible to start about 60 (idling) peers with a Quad-Core CPU and 12 GB RAM. Adding this to the spontaneous dropouts of the library, the emulation component is currently only usable in a limited scale. We remain with the hope that implementations stabilize after the WebRTC specification is finalized. Until then, bugs and incompatibilities hinder large-scale WebRTC emulation.

4.8 Demo Applications

During the work on the BOPlish core, we implemented several demo applications that assisted in showcasing our work. The applications are used to present the concepts of BOPlish in an accessible way. For us, this is an important aspect as developers shall be motivated to leverage BOPlish for their own applications.Moreover, the demos allow to prematurely find problems related to the user-facing BOPlish API for an overview on the API). The time fence of this work does not allow for very sophisticated applications but should give a glimpse into how a User Community operates. To separate the core BOPlish library from the application code, the demos reside in its own repository[17].

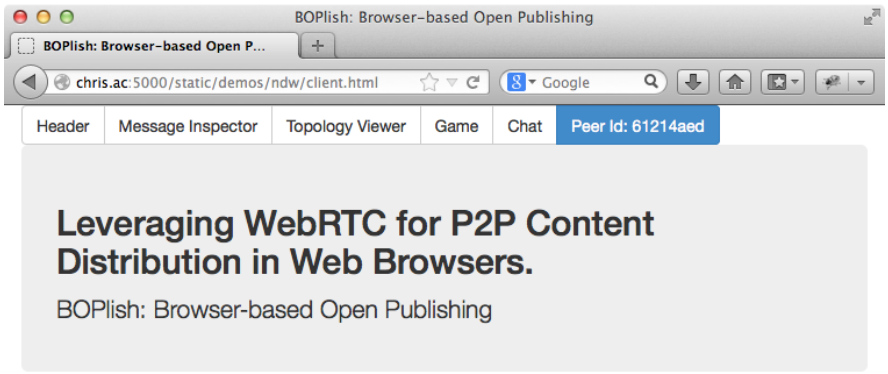


Figure 4.10: BOPlish demo client interface in a Firefox Browser

The demo applications use the Bootstrap CSS Framework[18] that allows for cross-browser Web frontend development. Even though all major browsers are capable of displaying the frontend, only Firefox and Chrome 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 Fig. 4.10).

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 Sec. 4.3.4).

[17] `https://github.com/boplish/demos`
[18] `http://getbootstrap.com`

4.8.1 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 Fig. 4.11) is separated into three columns. 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 4.4.1)
- *Bopcast Registration*: sends a `bopcast-protocol` register-request to the corresponding peer (see 4.4.1)

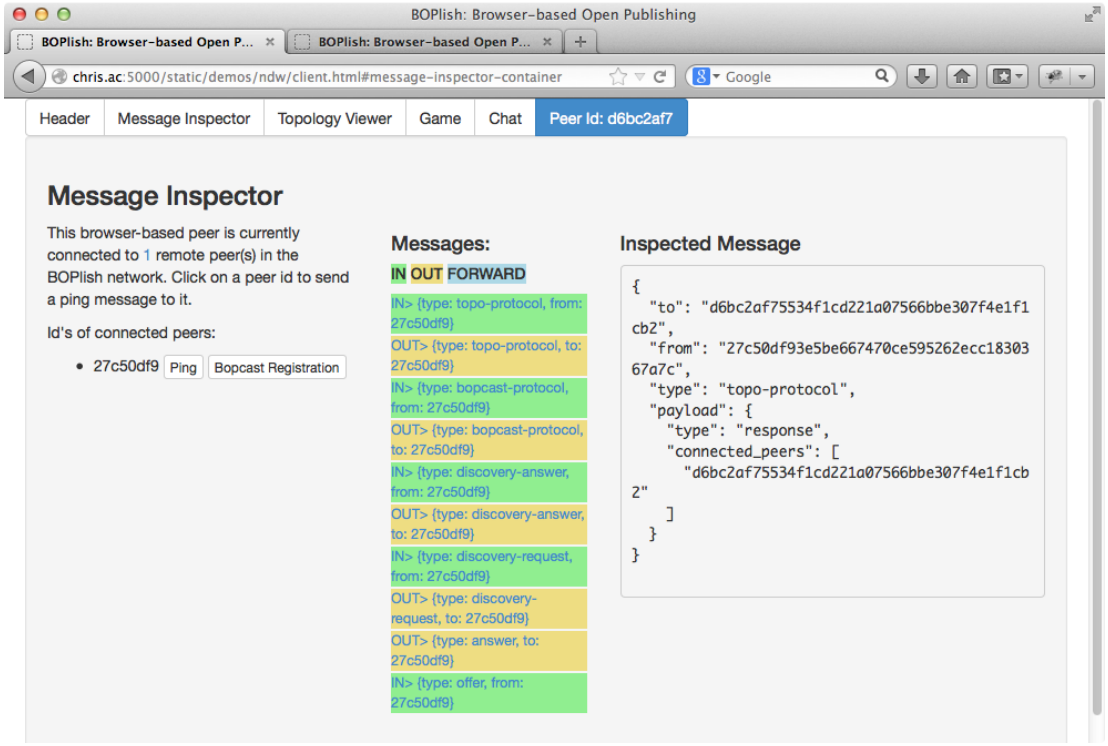


Figure 4.11: BOPlish Message Inspector demo application

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.

4.8.2 Topology Viewer

This demo application showcases the topology of a BOPlish User Community by displaying a graph of the network (see Fig. 4.12). 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.

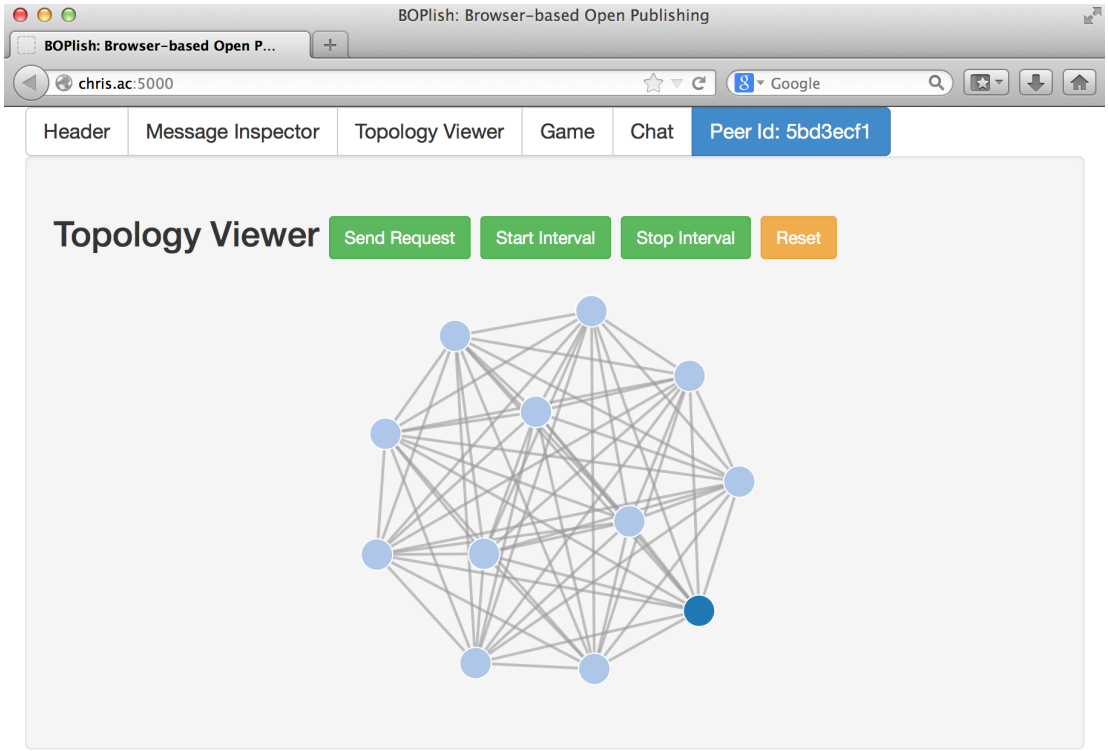


Figure 4.12: BOPlish Topology Viewer demo application

Buttons allow the user to interact with the `topo-protocol` that is used to gather the needed information (see Sec. 4.4.1). *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[19]. *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.

[19] `http://d3js.org/`

4.8.3 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 "player" on the black grid. All changes to the position are synchronized to the whole group. This application resembles our first attempt of an application leveraging BOPlish for group communication without any central server component.

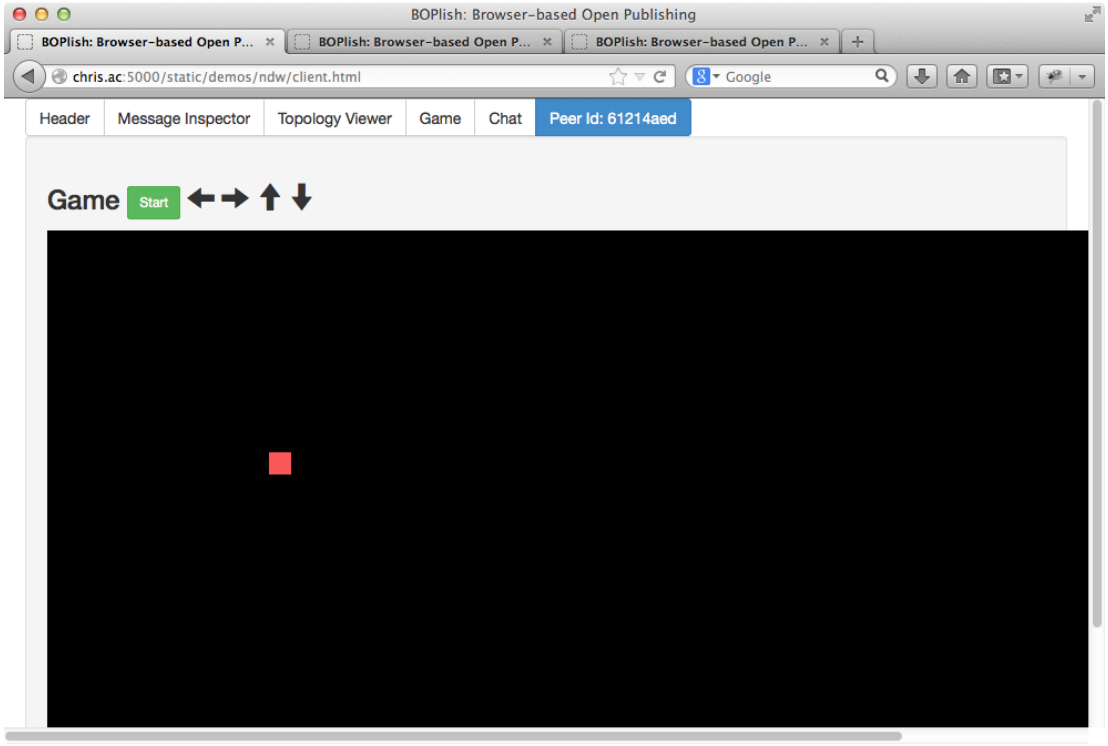


Figure 4.13: BOPlish Game demo application

4.8.4 Chat

The chat application allows BOPlish users to create chat rooms under the namespace of the user. Fig. 4.14 shows the administrative interface on the left and the actual chat communication interface on the right. Here, the user with BOPlish ID `dqbtvcqkwsi@id.com` creates a chat room called `star-trek-chat`. Two users join the room by using the generated BOPlish URI. After joining the room, messages can be published by sending them to the creator of the room. The creator keeps track of the memberships and relays messages to the other

users in the room.

Despite being a quite simple use case, this chat application is fundamentally different when compared to current applications on the Web. After the application has been downloaded from the Web server and the BOPlish User Community has been established, any user can act as a "server" for other peers, thus hosting an application for other users just by opening a browser window. The user that creates the room as well as the participants in the chat room do not in any way communicate with a server after the bootstrap procedure. Moreover, communication is naturally private as data is transmitted directly from one peer to another using end-to-end encrypted Data Channel.

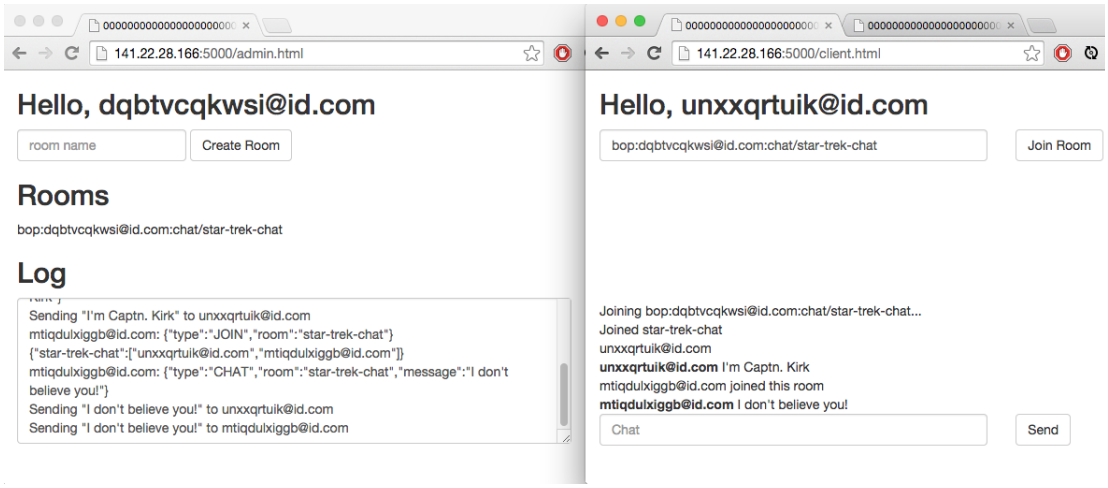


Figure 4.14: BOPlish Chat demo application

5 Evaluation

BOPlish is an approach to build user-centric applications on the Web. After laying out the included concepts and our implementation, as well as describing the peripheral components like the emulation environment, we can now continue to evaluate the system. To do so, we decided to take three perspectives with each covering a different angle of the evaluation. The perspectives are: Application, System and Security which we now define one-by-one.

In Sec. 5.1 we derive application contexts from typical use cases and thereby name examples of current Web applications. Using these applications, we first factor out important building blocks and examine how these can be mapped to BOPlish. Afterwards, composition rules showcase how a BOPlish application compares to the identified application contexts.

Afterwards, we conduct performance measurements using the emulation environment introduced in Sec. 5.2. This evaluation perspective shows response times and stability tests in both, normal and high load scenarios. Moreover, we measure bootstrap performance and compare it to current Web applications.

We end the section with the presentation of our security perspective in Sec. 5.3. Apart from a general WebRTC security assessment, it showcases new attack vectors resulting from the combination of WebRTC and a P2P layer directly in the browser.

5.1 Sketching BOPlish Applications

How do I implement my application on top of BOPlish? This is a valid question that arises after demonstrating the abstract capabilities of BOPlish. Obviously, the answer very much depends on the application characteristics that are to be implemented. We aim at partly answering this question by selecting examples of today's Web applications and sketching how these can be implemented on BOPlish.

In contrast to Sec. 4.8 where we presented simple demo applications and protocols to verify general functionality, we hereby give a glimpse at what application developers can expect from using BOPlish as compared to a centralized platform. This part of the evaluation is entirely theoretical as implementing the proposed applications goes beyond the constraints of this thesis.

To choose fitting applications, we recommence the use cases introduced in 3.1 that are initially inspired by current applications running on the Web. We choose two typical, widespread Web applications that fulfill these use cases, namely a file storage service that allows sharing between users (Dropbox) and a real-time chat (Facebook Chat) with group features. The applications are intentionally contrary in their usage and their imposed requirements to cover a greater range of functionality.

Dropbox is a Web-based, centralized platform where users upload arbitrary content which is then stored on and accessible from the centralized Dropbox server. The significant requirements for such applications lie in the availability of content as well as performance of file up/download. Real-time applications like the Facebook Chat, on the other hand, have a very different focus. Here, the focus is shifted towards a timely delivery of rather small messages, namely the user chat conversation.

5.1.1 Building Blocks

We break down the chosen applications into a list of building blocks. This list is not meant to be complete in the sense that it covers all functionality of the chosen application. Rather, it showcases important components, briefly outlines how these can be implemented in BOPlish and what restrictions apply.

URI-based Interaction

Web applications typically use URL identifiers to address objects in the system or provide access to specific functionality. Identifiers can be hidden behind UI components for improved user convenience. For example, a user may click a button that redirects the browser to a specific page or executes a predefined functionality. When a Dropbox user wants to share a file, a unique URL is created (Lst. 5.1 a)). Other users can get access simply by having knowledge of the identifier. Facebook also uses globally unique identifiers to address chat groups as shown in Lst. 5.1 b) but requires users to explicitly authorize themselves before gaining access to the chat room.

```
1 a) https://www.dropbox.com/s/[unique-id]/[file-name]
2 b) https://www.facebook.com/messages/conversation-[unique-id]
```

Listing 5.1: Examples of URL-usage from Facebook and Dropbox

BOPlish protocols can leverage the location- and application-independent identifiers to accomplish the same level of URI interaction. A file sharing application could provide a sharing functionality using identifiers derived from the application protocol in use. Lst. 5.2 a) shows

an identifier that might be used to call a `get` functionality on a specific file. It is also possible to create identifiers with unique ids as shown in Lst. 5.2 b). Addressing a chat room follows the same principle (Lst. 5.2 c)).

As such, BOPlish protocols are equally free in the design of identifiers as compared to current Web applications while still being location- and provider-independent.

```
1 a) bop:alice@example.org:file/get/[file-path[?parameters]]
2 b) bop:alice@example.org:file/get-unique/[unique-id[?
     parameters]]
3 c) bop:alice@example.org:chat/[chat-room-id]
```

Listing 5.2: Examples of URI-usage in BOPlish

Filesystem Content Access

JavaScript code running in the browser does not have direct access to the file system. Instead, user interaction is required for this task. This is reasonable because browsers run applications by simply visiting a web site instead of requiring the user to explicitly install a piece of software. The user can drag and drop the file to the browser window or the application might request a file upload window as an interface to the file system. Once the application gains access to the file it can read its contents and meta data just like a OS-level application can. Dropbox heavy relies on file system access and using the built-in browser technologies would be cumbersome for the user. Therefore, a OS-level client application is used that synchronizes files to the Dropbox back end.

BOPlish applications depending on data from the user file system underly the same significant restrictions as any Web application does. To gain convenient access to the file system, an OS-level application is required. BOPlish provides such a client through the Node.js-based headless runtime described in Sec. 4.7.1. Without such a client, each file access requires explicit user interaction.

Content Persistence

Dropbox and Facebook use a server-based, proprietary back end to persist content. Retrieval is only available for authenticated users. Browsers are currently not equipped with functionality to store large amounts of data. In-memory data is lost as soon as the user refreshes or navigates to another page. Small amounts of data (~ 5 MB) can be stored using the browser Web Storage interface (i.e., `localStorage`).

When a peer closes the browser or the tab that runs a BOPlish application, the WebRTC connection to the User Community is lost. As a result, the shared content from this peer is not available anymore. Content persistence cannot happen on central entities as there are none. A DHT is used to persist name resolution information by spreading content throughout the network. However, this mechanism is used for identifier resolution only and is not accessible to BOPlish application protocols. Persisting content beyond in-memory capacities or `LocalStorage` again requires the headless runtime. Moreover, content in BOPlish is by default only accessible when the user is actually available in the network. To keep content highly available, a user may want to run a stationary device (such as a home NAS server) that is always connected to the User Community.

Real-time Interaction

Real-time Interaction happens when two end points communicate directly, as occuring in a Facebook chat where users expect fast message delivery times (typically in the range of a few $100ms$). Facebook uses a central server to relay messages between users. Proxying the messages over a server obviously results in higher delays as compared to connecting the communication endpoints directly.

BOPlish directly connects users and therefore allows for lower end-to-end delays compared to current centralized Web applications. The SCTP transport allows for high-bandwidth and low-latency connections. Moreover, the server is relieved from the load imposed by routing messages. Due to the underlying P2P paradigm, central control is not easily possible.

Group Interaction

Facebook users can create, join and leave chat groups. Messages in the groups are disseminated in a broadcast manor. Due to the centralized server that aggregates all messages, group maintenance and message distribution is trivial.

In BOPlish, group interaction in small User Communities can be implemented by directly connecting all participating peers to a full mesh (e.g., by using the bopcast protocol introduced in Sec. 4.4.1). Once the group size increases, the full-mesh approach does not scale. We leave this question open for now and dedicate Sec. 6 for an in-depth discussion of a group communication mechanism for BOPlish applications.

5.1.2 Application Composition Résumé

We discussed main building blocks that make up the Dropbox file-sharing and Facebook group chat applications while pointing out the distinctions when compared to a BOPlish approach. By implementing and combining these building blocks in BOPlish, applications can be composed.

We now factor out peculiarities occurring in BOPlish-based applications that arise when combining the building blocks to give the reader a deeper understanding of how BOPlish applications have to be designed. This includes routing policies that are specific to the P2P layer and mobility capabilities for peers changing location that deeply differ from the current Web.

Content transfer in BOPlish is encrypted end-to-end due to the underlying DTLS connection between the endpoints. This is something that centralized Web services typically cannot offer because the application itself runs on the central Web server and requires access to the unencrypted content. In BOPlish, peers stay in full control over the content. As such, applications trying to provide a certain level of privacy for their users benefit from the P2P paradigm used in BOPlish.

Current centralized Web platforms face attacks from malicious crackers and lawsuits from governmental institutions. Once gained access, all users are endangered. In BOPlish, on the other hand, particular peers have to be attacked, making it harder to perform attacks on many users.

Another differentiating factor in BOPlish is the binding to user identity instead of location. BOPlish inherently provides mobility functionality by tying any content to a user-centric instead of a location-centric identifier. When a host changes location, the name-resolution mechanism keeps track of this change by updating the corresponding DHT value. The protocol itself does not have to handle a location change as the user identifier stays intact even if the IP address changes. Current Web applications are typically running in fixed locations and changing the location requires updating the DNS entry.

After presenting how a BOPlish approach of typical use cases compares to the traditional understanding of Web application, we conclude that some of the typical building blocks have to be implemented very differently. Not every use case can be easily implemented, content persistence requires a stable connection to the User Community and a headless runtime because the browser environment is not yet ready for such usages. Improved privacy is a plus for BOPlish applications due to the missing central content aggregation, as well as the URI scheme in use that releases of the tight coupling of content and service provider.

5.2 System Performance Evaluation

We now continue to test the system for its functionality using the emulation component introduced in Sec. 4.7. Currently, existing WebRTC implementations are neither feature complete nor do the performance characteristics match the finalized product. For example, the Chrome implementation currently does not allow setting options for the SCTP stream that is used for the Data Channel connections. As such, only reliable transmission can be tested. As a result, we did not observe any packet loss during the tests at all. Another limiting factor is the number of open Peer Connections a browser can cope with. Again depending on the implementation, we observed that number range from 8 (Android smartphone with Firefox) to 30 (PC with Chrome x64).

Because of the implementation differences and the prematurity of some parts of WebRTC, we do not give detailed insight on performance characteristics at this point as they are likely to change. We rather test our system for functionality and conceptional correctness.

User Communities rely on a DHT for name resolution purposes. It is the main bottleneck for performance as any content distribution only occurs one-to-one between the peers. In BOPlish, we implemented the Chord DHT but other protocols like Pastry or CAN are certainly possible, too. The logarithmic scalability of DHTs allows for large numbers of peers even under the imposed implementation restrictions (limited number of PeerConnections per browser). In our tests, we compare one-hop as well as two-hop and 10-hop performance. The one- and two-hop measurements shall give an idea of the scaling behavior when compared to a 10-hop routing. Chord uses $\mathcal{O}(\frac{1}{2}\ log_2(N))$ hops in average to route to a key. 10 hops thus equal about 10^6 nodes in the system with a fully populated finger table:

$$\begin{aligned} 1/2 * log_2(N) &= 10 \\ log_2(N) &= 20 \\ N &= 2^{20} \approx 10^6 \end{aligned}$$

We do not expect such large group sizes but our implementation uses a dynamically sized finger table that depends on the number of Peer Connections the respective browser can handle. A smaller finger table leads to a greater hop count. Still, 10-hops map to long distances even with small finger tables and should give a realistic upper bound. As such, we chose it as a reference value.

5.2.1 Configuration

To evaluate BOPlish, we used a total of 4 hosts. All machines use Intel QuadCore CPU with 2,33 GHz to 3 GHz and a total of 40 GB RAM. One of the hosts runs both, the bootstrap server and the emulation mediator. The three other hosts start an instance of the emulation host instance that in turn spawns BOPlish peers. During the measurements, the emulation hosts were monitored for CPU and RAM usage which did not exceed 80% at any time.

The hosts are interconnected using Gigabit Ethernet (GigE) and use public IP addresses. The public addresses turned out to be a problem with the STUN implementation of the Chrome Browser as it apparently expects to operate in a NATed environment (this is a bug). With the current implementation the initial STUN connection establishment takes a long time (about 10 seconds), significantly distorting the delay measurements. As a work around, we used a locally started STUN server[1].

Our emulation configuration does not reflect the actual Internet environment. Delays will be higher in the wild due to longer routes and bandwidth between peers will be way below the GigE speed we achieve in our test setup. As a future work, we plan to deploy the emulation environment to an environment which more closely reflects the Internet like PlanetLab[2].

For our tests, we disabled BOPlish lookup caching to not disturb the analysis. This mechanism caches resolved identifiers such that subsequent queries to that user identity can be served locally instead of again resolving it through the DHT. Thus, every request to a BOPlish ID gets resolved to a corresponding peer ID prior to the actual content transmission.

Achieving 10 hops in a Chord DHT would require a huge number of nodes. Our computing capabilities as well as the central emulation mediator are not geared up for that task. Instead, we decided to alter the finger tables of the peers so that every peer only knows its direct successor and predecessor. Thus, every request strictly follows the ring topology constructed by the Chord algorithm.

5.2.2 DHT Stability Boundaries

Our first test is supposed to be a smoke-test. We want to find the upper bound of message per second the DHT can cope with. We learned that the results differ depending on the WebRTC implementation respectively the browser we used in our tests. Lst. 5.3 shows the code that has been used to conduct the measurements.

[1] https://launchpad.net/ubuntu/trusty/+package/stun
[2] https://www.planet-lab.org/

```
var hostBopId = 'drnlbbnzs@id.com',
   start = new Date(),
   j = i = 1000;
while (i--) { // send i messages to bop id
 bopclient._get(hostBopId, function(err, msg) {
   if (!--j) calculate(start); // wait for j callbacks
 });
}
function calculate(start) {
 var took = new Date() - start; // [ms]
 var msgPerSec = 1000*1000/took; // normalize to msg/sec
 console.log('msg/sec:', msgPerSec);
}
```

Listing 5.3: BOPlish DHT performance testing code

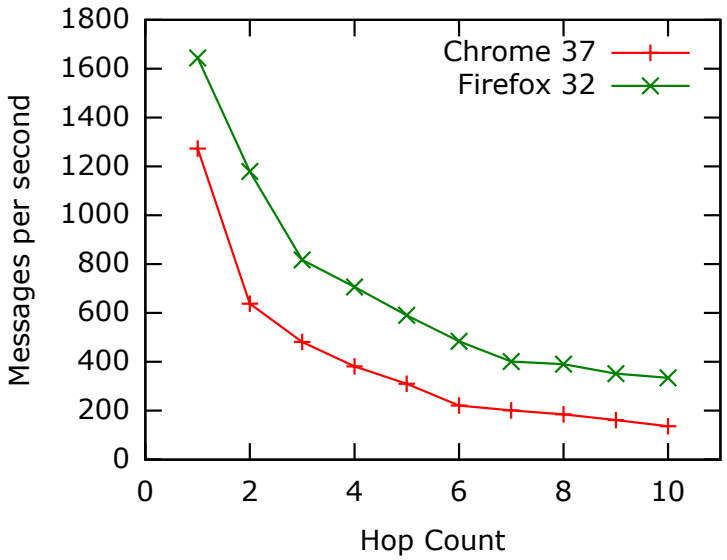


Figure 5.1: BOPlish DHT performance behaves differently according to the underlying Web browser

The results are shown in Fig. 5.1. Chrome one-hop performance tops at 1270 $\frac{msg}{sec}$ and declines to 130 $\frac{msg}{sec}$ when the messages are routed over ten hops. Firefox performance is superior throughout the test and it is able to push 1640 $\frac{msg}{sec}$ through the BOPlish infrastructure. 10-hop performance is more than doubled compared to Chrome with 330 $\frac{msg}{sec}$. Even though we saturated the number of messages that the implementation can handle, we experienced no

break downs. During the processing, the Web application was unresponsive and thus unusable. Though, it came back to a stable state once all messages were processed.

It has to be noted that other peers might mark a peer as failed when it is stuck processing messages. The proactive DHT maintenance mechanism that periodically checks neighbors would then issue the DHT maintenance mechanism, leading to further increased DHT load. Application developers might want to adjust the periodic timeout for failure detection to accommodate to the message load or use some sort of multi-hop flow control.

The results leave us confident that our solution can sustain even high amounts of messages and does not break down in case of overloading. We hope to see increasing stability with maturing implementations.

5.2.3 Bootstrap Delay

To offer competitive user experience, applications that rely on BOPlish must be usable as soon as possible after the initial Web application has been loaded. A crucial factor to minimize the time between page load and application initialization on the client is the bootstrap delay. This factor depends mainly on the time for the new peer to join the DHT, i.e., initialize finger tables, find its place in the Chord ring and update successor and predecessor. To measure the delays, we had to intervene into the insides of BOPlish, making it impractical to show the actual code we used.

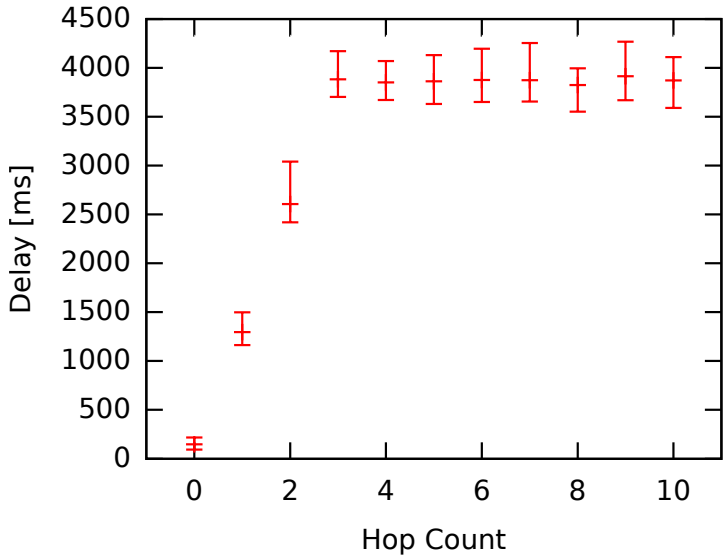


Figure 5.2: Gross BOPlish bootstrapping performance

Fig. 5.2 outlines the bootstrap delay in the environment described above and indicates the total time from instantiating the BOPlish client object until the peer has joined the DHT. We conducted the numbers by subsequently joining additional peers, thus increasing the hop count that is necessary to route messages from one peer to another. We expected a linear trend from the results but the results show a different picture. They can roughly be divided into two stages: The first stage is the one where only two peers have joined the DHT, the second stage is the one displaying the delay with three and more hops on the x-axis.

It can be seen that, in the second stage, the maximum delay for bootstrapping remains more or less constant with increasing hop count which seems odd. The grave reason for our results are the very high delays introduced by the WebRTC connection establishment. The connection establishment delay is so high that all other operations on the resulting Data Channels (the routing of the messages) are negligible with regards to delay measurements. We therefore conducted a second measurement that leaves out WebRTC connection establishment: the net bootstrap delay.

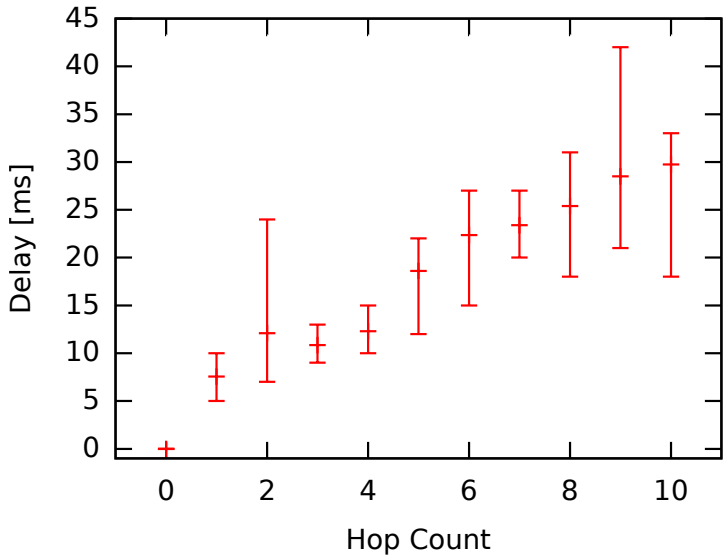


Figure 5.3: Net BOPlish bootstrapping performance shows near-linear behavior with increasing hop counts

These numbers provide for a better evaluation of our actual implementation. Fig. 5.3 displays the time from instantiating the BOPlish client until the join operation has succeeded as a function of the hop count used to route the join messages. Here the bootstrap delay increases with more peers joining the DHT, just as we expected.

5.2.4 DHT Lookup Performance

We continue our evaluation by determining average lookup delays that occur when issuing GET requests against the DHT under little load. Such a lookup closely resembles the name resolving procedure. The code is shown in Lst. 5.4. Due to the stable, low-delay LAN environment, the majority of the measured results mirror the actual delay introduced by the BOPlish overlay and the WebRTC stack.

```
var counter = 100,
 getDelays = [],
 hostBopId = 'bwjllzqiitpgb@id.com';
(function () {
 var timeStart = new Date();
 bopclient._get(hostBopId, function(err, msg) {
  getDelays.push(new Date() - timeStart);
  if (--counter) getDelay();
  else calculate(getDelays);
 });
})();
function calculate(values) {
 var min = Math.min.apply(null, values);
 var max = Math.max.apply(null, values);
 var sum = values.reduce(function(pv, cv) {
  return pv+cv;
 }, 0);
 var avg = sum / values.length;
 console.log('avg, min, max:', avg, min, max);
}
```

Listing 5.4: DHT Lookup Delay

Fig. 5.4 shows the results of our tests. One-hop performance shows a minimum of $4\ ms$ and a maximum of $43\ ms$. As expected, two-hop delays are roughly doubled. 10-hop delays show a big span between minimum and maximum values. This result might seem unexpected in a stable LAN environment but can be explained by the heavy-weight WebRTC stack (the browser runs a full-fledged SCTP and DTLS stack) and the multitude of components working together.

Overall, we are satisfied with the results as the added delay from the name resolving mechanism is not very high and mostly ranges below $50\ ms$. Moreover, we expect the average delays to decrease when WebRTC implementations mature over time.

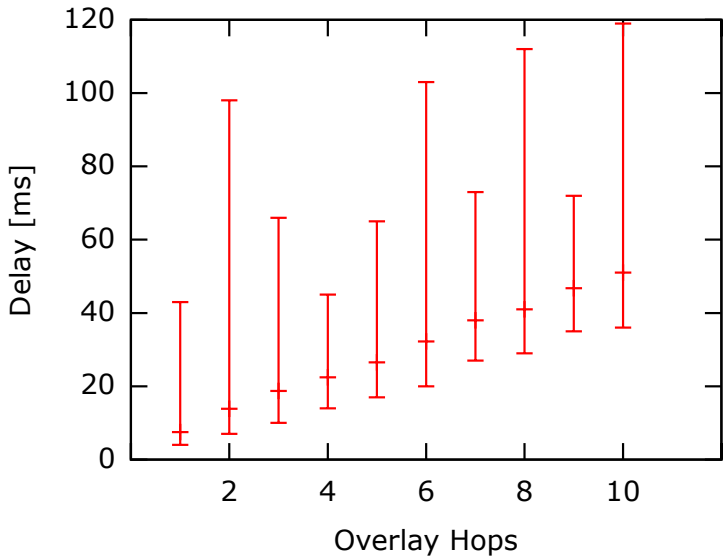


Figure 5.4: BOPlish Lookup Performance

5.3 Security Analysis and Attack Scenarios

The combination of WebRTC and a Web-based P2P network potentially creates new attack vectors. Since one goal of BOPlish is to improve users privacy in communication scenarios, we focus especially on attacks on privacy. We outline the general attack surface, define security objectives and give an overview over selected attack scenarios.

5.3.1 Attack Surface

The attack surface of applications or protocols describes the various ways an attacker could circumvent the security measures of a system. Every input/output (e.g., a user interface or data transport channel) of an application is a potential entry point for malicious activity, either specifically harming targeted users or system stability as a whole. Today's Web applications typically have a well-known attack surface and can be analyzed systematically. The Open Web Application Security Project (OWASP) offers insights into such analysis[3]. Classic P2P systems that are based on standalone applications on top of UDP or TCP have also been broadly studied. Sit and Morris provide a general overview and categorization of attacks on DHTs in [84].

The combination of Web applications and P2P functionality (via WebRTC), though, creates a completely new playing field, possibly opening up applications to as-yet-unknown attacks. To model the attack surface of P2P Web applications (such as BOPlish), we categorize malicious

[3] `https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet`

entities as either insiders or outsiders. Insiders are attackers that act as legitimate, authorized members of the P2P community whereas outsiders harm the system by, e.g., eavesdropping on wireless communication or injecting malicious messages into the underlying network infrastructure. The degree to which an attacker may carry out certain attacks heavily depends on the resources available. A single attacker with only access to a private laptop connected to the Internet may, e.g., not be able to intercept arbitrary network communication. On the other hand, government agencies such as the NSA have deep access into Internet infrastructure and can thus carry out much more sophisticated attacks. As such, the countermeasures heavily depend on the amount of resources available.

Fig. 5.5 illustrates the components involved in a WebRTC connection. We categorized these into trustable (green), possibly-trustable (yellow) and untrustable (red) components. Components such as the browser and the hardware are assumed to be trustable in our model. These presumptions narrow down the attack vectors to be analyzed. The Web application running inside the browser as well as the Web server delivering it are deemed possibly-trusted. This means that there may be applications/servers that are malicious and others that can be trusted. The current Web architecture has measures in place to decide on the trustability of applications and servers. These include server authenticity via TLS certificates, runtime sandboxes for JavaScript code and Same-origin Policy (SOP). The same assumptions apply to the identity provider. Other peers in the WebRTC communication model are per default not trustable. Every application (and thus, every programmer) must put in place mechanisms that help users decide whether the remote peer is malicious or not. This is a major difference to the attack surface of ordinary Web applications.

5.3.2 Security Objectives

Our analysis of WebRTC security and impacts on P2P networks is based on a set of safety objectives generally known as the CIA triad. The acronym CIA denotes the three objectives Confidentiality, Integrity and Availability. The semantics of these three goals explained below are inspired by [83].

Confidentiality

This objective ensures that data which is transferred between two trusted peers does not reach an unauthorized third actor. An example of the need for confidentiality is the submission of credit card information from a buyer to a merchant. In an Internet system the credit card number, name of the holder, expiration date, possibly a CVC check number and additional

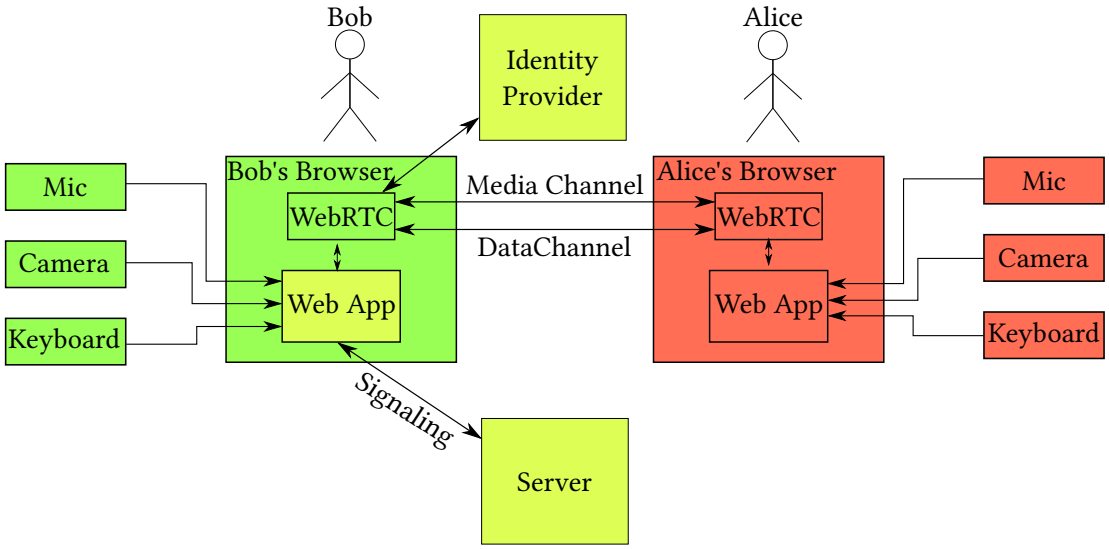


Figure 5.5: The WebRTC security architecture. Green color indicates entities that the user can trust if all specifications are implemented correctly. Yellow components can probably be trusted by user choice (e.g., depending on the origin domain). All other components of a PeerConnection are by default untrusted.

information usually pass a number of routers that lie between the buyer and the merchant. To make sure that no one on this way (e.g., with access to the intermediate routers) is able to read out the sensitive data it is encrypted and thus confidentiality is put in place.

Integrity

Data integrity in an IT infrastructure makes sure that data originating from one node is not modified on the way to the receiver or the receiver is enabled to check whether data has been modified. Referring to the credit card example above this means that no person with access to intermediate routers can inject false credit card information. Since (deliberate or unintentional) modification of data in IP networks cannot be avoided, one has to make sure that modifications are detected by the peers. In this scenario, the integrity of the data en route is ensured using message authentication codes (MAC) and cryptographic signatures.

Additionally, integrity of data refers to the unmodifiability of data stored in an IT system. This can also be ensured using cryptographic signatures or checksums by using hash algorithms like MD5, SHA-1/SHA-2/SHA-3 or RIPEMD and comparing the results. This is typically done when downloading files from a server.

As a part of integrity protection, the objective of authenticity implies the process of verifying

the claim that data coming from a certain origin actually originated there. This is important in various contexts: A web server sending HTML to a browser via HTTPS authenticates itself using a TLS certificate that is cryptographically signed by a trusted certificate authority. On the other side a user authenticates to a server using her credentials (e.g., user name and password or TLS client certificate). More generally authentication may be applied by something a peer knows (e.g., a password), owns (e.g., a key card) or is (e.g., a biometric attribute).

Again referring to the example of a customer buying goods in an online shopping system the client authenticates itself to the shop provider by logging into the system. This ensures that the purchase can securely be tracked back to the customer.

There are, however, use cases where authenticity – in the sense of being able to track information down to a real person – is explicitly not desired. These include anonymous/pseudonymous conversations between two peers to maintain a certain level of privacy.

Availability

The objective of availability guarantees that information stored in an IT system is accessible when needed. In the context of P2P systems it has to be assured that, e.g., denial-of-service attacks against certain peers do not lead to users not being able to access a certain piece of information. It is especially important in the design of P2P systems to guarantee availability since users may store content on peers that they do not control themselves.

5.3.3 General WebRTC Security Assessment

The IETF Rtcweb Working Group has devoted (and still is devoting) a vast amount of discussion on the security of WebRTC entities. Thus, the specifications, as a result of the discussions, include numerous security goals such as confidentiality, source authentication and identity assurance. The security aspects of WebRTC are specifically identified in [72] and further outlined in [73]. An overview of the trust architecture implied by WebRTC is given in Fig. 5.5. The following sections provide insight into how each entity in WebRTC is supposed to comply with our security objectives.

Server

Delivery of the application is done in the classical way via HTTP or HTTPS and thus the same security considerations that apply to every Web application delivery also apply here. Therefore, these are not WebRTC-specific and not further investigated. Implications derived from the fact that using a central server may expose certain meta data about the users – which can be

a threat to privacy – are outlined below. One important aspect to consider when deploying the server is that delivery of the application can be a first entry for attackers, e.g., when the transfer is conducted via unencrypted HTTP. If this is the case, a man in the middle may be able to introduce malicious code to the user and perhaps fake a WebRTC connection. Thus, application transfer should always be conducted using HTTPS with proper server certificates.

Browser

The WebRTC specification demands several measures from browser implementors in order to guarantee users' security, as stated in Sec. 2.3.2. Still, especially in P2P scenarios, it may be possible that random users start connecting to the client's browser and other peers may get access to the client's IP address which poses threats to location privacy. Since any Web application (malicious or not) a user points her browser to may be able to initiate WebRTC connections, the browser must also restrict the applications' access to certain information. The WebRTC Security Architecture specification [73] mentions several countermeasures to these threats which are elaborated on in Sec. 2.3.2.

Currently, a consent is granted for the whole site and not individual incoming calls. This may be convenient for the user but poses the threat that an arbitrary user uses the calling service to call any other user currently connected to the service. One way to handle this would be that the application code asks for consent when an incoming call is to be accepted. In a VoIP application for example the user may be prompted by the application that another user is calling. Since this is left to the application there may be applications that leave users open to being tapped by other users. It shall be noted that the indicators are only used for audio/video connections and not for WebRTC Data Channels. For the latter, there exists no consent mechanism for outgoing or incoming connections.

The security drafts [72, 73] both contain a dedicated section dealing with location privacy concerns. These arise when negotiating Interactive Connectivity Establishment (ICE) parameters between two browsers prior to establishing a WebRTC connection which may leave the user open to revealing her IP address to another peer without her having ways to suppress this. The specification documents therefore mandate implementations to supply JavaScript applications with a means to suppress ICE negotiation until the user has explicitly granted the connection initialization. Guaranteeing location privacy hence is a task left up to any individual application.

Identity

Handling user identities in the WebRTC context is discussed in detail in [73]. With regards to the authenticity safety objective mentioned in Section 5.3.2 there are two general concepts available to deal with identities:

1. Anonymity and Pseudonymity
2. Identity Providers

Anonymity and pseudonymity are useful when it is not desirable or necessary to identify a peer. An example mentioned in the security architecture specification is that of a "click to call support" button on a company's website. Here, the company's call center agent must not necessarily know the real identity of the caller only to deal with general product questions.

Identity providers are outlined in [73] as third entities that mutually ensure users' identities so that every user can be guaranteed that the identity she claims to obtain is proved by a trusted third party (see Fig. 5.6). This implies that user A trusts the identity provider of user B to securely prove her identity.

In detail, an application may ask an identity provider to generate a cryptographically secured identity assertion that is then carried over the signaling layer together with the offer/answer packages needed for connection establishment. Such an assertion is then extracted from the package by the peer on the other side and sent to the identity provider for validation.

A current concern with the identity provider approach laid out in the specification is that no browser has implemented even parts of that mechanism. Thus, usable implementations may come to users rather late in the WebRTC rollout process. On top of that, Identity assertions are only available for audio/video channels and not for Data Channels.

5.3.4 Attack Scenarios

We now examine different attack vectors based on four general scenarios and outline possible countermeasures that are to be put in place by WebRTC P2P application developers. A current shortcoming of browsers are missing functionality in public crypto. E.g., browsers lack signature verification or encryption using an asymmetric key pair. The W3C is working on an API to make these available to Web applications [23]. In the meantime, the missing functionality can be included by incorporating JavaScript implementations of cryptography functions into BOPlish. However, these cannot provide the same security as a native API because browsers lack access to, e.g., a cryptographically secure random number generator. All the scenarios

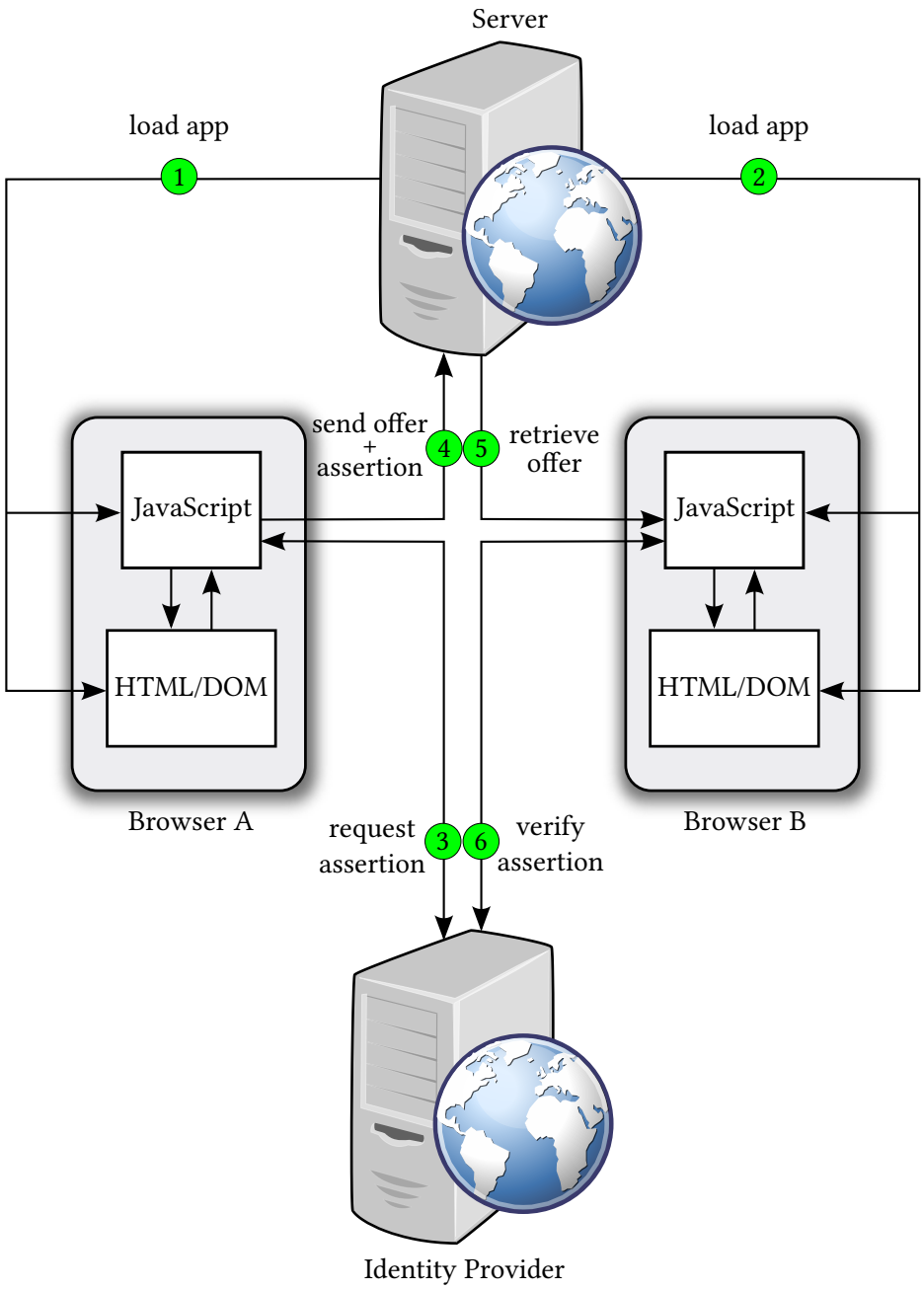


Figure 5.6: WebRTC identity providers (IdP) allow for the verification of remote peers' identity. Here, two browsers A and B load an application (steps 1 and 2). Then browser A requests an identity assertion from its IdP (where the user has already authenticated) (step 3) and sends that assertion together with the WebRTC offer to browser B (steps 4 and 5). In step 6, browser B asks the IdP of browser A to verify the assertion and identify the user of browser A.

mentioned below as well as possible countermeasures thus assume such functionality as being present.

Outside Monitoring Attack

WebRTC-based Content Communities are potential targets of pervasive as well as targeted monitoring. Pervasive monitoring is the act of secretly (i.e., without the agreement of the participating parties) recording a communication. The surveillance is not restricted to specific targets but tries to span the network as a whole. The IETF has just recently acknowledged pervasive monitoring as a first-class attack that should be mitigated [31]. In contrast, targeted monitoring is specific to single hosts, e.g. a peer in the P2P network or the signaling server. Our attacker in this scenario looks like this: Monitoring is conducted strictly passive, meaning that no data is altered, dropped or inserted in the monitoring process. Due to the characteristics of the WebRTC technology, data and signaling transmissions are handled separately. The attacker is able to monitor packets of a WebRTC application on layer 3 and thus has access to a router on the path between peers or between the signaling server and peers.

As the Data Channel specification requires DTLS for all data connections, the transferred data between peers can be considered safe from eavesdroppers that do not have access to the DTLS keys (which are negotiated via Diffie Hellman key exchange and thus are not transferred). Nevertheless, meta data can be collected and evaluated. When users connect end-to-end, the IP headers reveal the communicating endpoints. Hop-by-hop routing can be used to disguise network paths and therefore hamper passive monitoring. TOR, for example, uses onion routing to hide the origin of a request[4]. In general, the routing mechanism of the P2P overlay dictates the level of obfuscation.

In order to specifically monitor a WebRTC-based network, it is worthwhile to target the signaling path. The signaling messages contain valuable information (Lst. 5.5) that allow an attacker to learn about the network topology. The WebRTC specification does not dictate the type of channel to use for transmitting signaling messages. As such, implementers might choose to use an unencrypted channel such as the non-TLS variant of WebSockets or HTTP. While not directly apparent, this puts users privacy at risk when traffic monitoring is in use.

The vast majority of current WebRTC applications share a central component: a server that handles signaling between the users. Even if the signaling channel is encrypted, analyzing incoming and outgoing packet header could allow an attacker to effectively gather information about the network topology from a central vantage point. An approach to mitigate this problem is in-band signaling. If a P2P overlay has already been established between the users, the

[4] http://www.onion-router.net/

overlay can also be used to route signaling messages without the need for a central server. This drastically reduces the possibilities to pervasively monitor traffic in the P2P network, which is a major benefit of using a WebRTC P2P network for signaling. The next attack scenario, though, focuses on monitoring attacks conducted by malicious peers inside of the P2P layer.

Inside Monitoring Attack

In pure structured or unstructured P2P systems the P2P layer is used to route traffic through a number of peers between originator and destination. Thus, a peer may monitor all the traffic that passes through it. Based on that traffic, malicious entities may be able to profile certain users' behavior and read unencrypted, possibly sensitive data. Adding to this the possibility of Sybil attacks (i.e., one entity joining the network with a huge amount of identities, each acting as a single peer) an attacker may be able to monitor complete traffic flows between peers. By creating enough Sybil peers, an attacker has the ability to control most of the traffic and effectively influence the availability of the stored data.

We can assume that it is very hard to counter Sybil attacks without a central authority [27], even though concepts exist to counter them by, e.g., employing social networks [106]. Knowing that, we only have the option to uncover as little information to intermediate peers as possible. For routing traffic in a P2P network there is certain meta information every routing peer needs to be able to read, such as the address of the destination peer. In a DHT this is most possibly the only information that intermediates need to access. All the rest (meta information as well as payload) may be hidden from them and only be readable by the recipient. A form of end-to-end encryption on this data may be applied in this case.

Coming back to the Web context of WebRTC applications this poses a specific problem: There are still no standardized encryption APIs in today's browsers. As a result, applications using WebRTC for P2P networks would have to rely on third-party JavaScript libraries[5] or browser add-ons[6].

An additional countermeasure would be to apply a kind of onion routing[7] so that the intermediate peers only know their direct neighbors when routing messages. This mechanism hides the original sender and receiver from intermediates. More advanced techniques such as trust-based routing [39] are also applicable but add to the complexity of the routing algorithm.

[5] https://github.com/bitwiseshiftleft/sjcl
[6] https://github.com/polycrypt/foxycrypt
[7] http://www.onion-router.net/

Redirection Attack

Instead of only passively monitoring the transmission, an attacker can also try to alter, drop or inject messages to induce a specific behavior of the application. Here, we have two different attacker models: An outside attacker not taking part in the DHT and an inside attacker that acts as a legitimate peer in the DHT. While the DTLS-protected Data Channel is naturally secured against tampering attempts by outsiders, insiders can easily modify SDP messages routed through the DHT. An insider that receives an SDP message for forwarding can change the candidates contained in the SDP (Lst. 5.5) to e.g. make the remote peer establish a connection to the attacker peer.

For outsiders, the signaling channel is the only target for this type of attack (assuming that the attacker has no resources to crack the DTLS encryption between peers). If the attacker is able to alter signaling messages, man-in-the-middle attacks are relatively easy to perform and take the same form as insider redirection attacks. The attacker could alter the SDP information such that both systems establish a Data Channel connection to him instead of each other. He could then read and/or alter all content passed between the users.

```
1 v=0
2 o=Mozilla-SIPUA-29.0.1 1097 0 IN IP4 0.0.0.0
3 s=SIP Call
4 t=0 0
5 a=ice-ufrag:d9be6af0
6 a=ice-pwd:62fef8346f4b35cf0c9d073e41e52f27
7 a=fingerprint:sha-256 <omitted>
8 m=application 56116 DTLS/SCTP 5000
9 c=IN IP4 10.17.32.10
10 a=sctpmap:5000 webrtc-datachannel 16
11 a=setup:actpass
12 a=candidate:0 1 UDP 2130379007 10.17.32.10 56116 typ host
13 a=candidate:1 1 UDP 1694236671 93.198.223.40 56116 typ srflx
     raddr 10.17.32.10 rport 56116
14 a=candidate:1 2 UDP 1694236670 93.198.223.40 60571 typ srflx
     raddr 10.17.32.10 rport 60571
```

Listing 5.5: Offer information (SDP) generated by a Firefox browser

Write access to the signaling information is thus an effective way to get access to the transmitted content. We can identify multiple attack vectors that can lead to such man-in-the-middle attacks. Again, using a centralized server for signaling can be risky. Gaining control over such a server (e.g. through a system vulnerability) would compromise the whole

system. It is also assumable that an outside attacker redirects the signaling messages to a server under his control. This could be achieved with, e.g., a DNS redirection attack where the DNS falsely returns wrong IP addresses. Here, the attacker model assumes a quite powerful attacker, capable of maliciously changing DNS entries (e.g. a government agency having access to ISP-level DNS servers). As a result, the attacker redirects the signaling information to a server under his control.

As WebRTC clients are expected to operate behind NATs, STUN is used to determine a path through which the client is reachable. The STUN response is embedded in the signaling information (see *candidate* lines in Lst. 5.5). A vicious STUN server could be used to fake these candidates and reply with an address of a system under the attackers control. This, again, assumes a powerful attacker, having access to resources to modify the behavior of the STUN server in use.

Remote Code Execution Attack

This attack stems from the fact that a Web application is combined with P2P capabilities: Here, the attacker model assumes an attacker acting as a legitimate peer in the P2P network. Additionally, the application is developed in the way that it does not treat every input coming from remote peers to be potentially vulnerable. This opens the door for an attacker to execute JavaScript code on other peers. In BOPlish, messages are exchanged in JSON format and encoded for transfer via `JSON.toString()` and decoded on the receiving side via `JSON.parse()`. If, for some reason, an application would treat part of the JSON object as executable, e.g. via `eval()`[8] calls, the peer is open to a remote code execution attack. Research on the feasibility of attacks on WebRTC-based applications with a focus on P2P networks is already being conducted. Gallersdörfer, for example, examines ways to hijack applications and convert browsers them into peers of a browser-based botnet [34].

The outlined scenarios demonstrate that there is potential for new attack vectors in WebRTC-based P2P networks. Application developers, even those making use of BOPlish for handling network creation and maintenance, will have to acknowledge these new vectors in their security assessment. On the other hand, WebRTC P2P networks can offer better privacy, especially when it comes to protecting against pervasive or targeted monitoring attacks. Further research on the possibilities to maliciously make use of such networks has to be conducted to categorize those new types of attacks and provide structured guidance, such as OWASP has conducted for classic Web applications.

[8] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`

6 Group Communication

BOPlish is our approach to a generic, user-centric publishing facility on the Web. Peers are loosely coupled in a structured overlay and a name resolution indirection allows to dynamically map identifiers to peers that are currently available. The implementation presented until now is able to exchange data between peers in a one-to-one fashion. A natural functionality that arises in many possible application scenarios is a group communication mechanism to allow efficient many-to-many communication within User Communities. Such a mechanism would be a valuable addition to the BOPlish ecosystem.

One example that showcases the need for such mechanisms is the group chat application introduced in Sec. 4.8. In our current solution, data is transferred point-to-point between two peers. While the name resolution is always using the DHT, data distribution occurs in a full mesh between the participating clients (i.e., the group chat members) using the `bopcast` protocol. Groups are bound to small sizes due to limited scalability of the full mesh data distribution and browser constraints (due to the limited number of Peer Connections). We aim at mitigating the imposed limitations by providing a mechanism that scales well beyond the current limits of the full mesh.

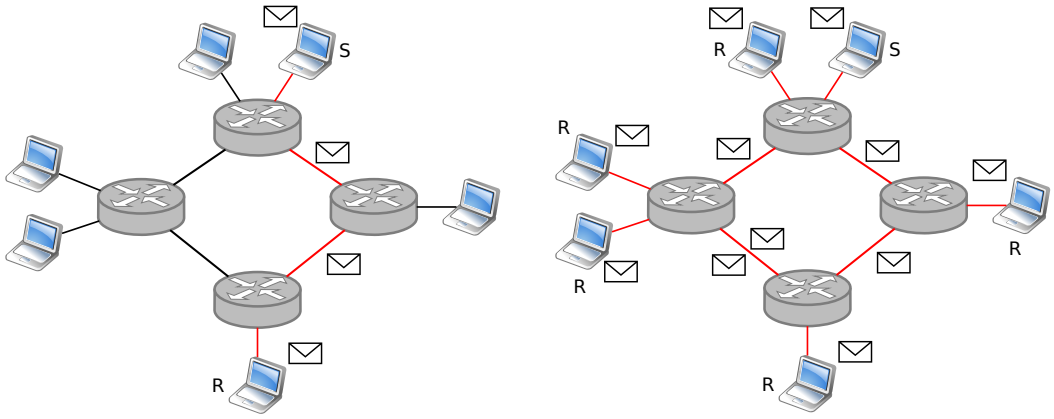
Multicast fosters such group communication mechanism but efficient IP-layer multicast suffers from deployment issues in the public Internet. Therefore, many ALM systems have been introduced that promise high bandwidth, low delay characteristics while being deployable on today's Internet. While this seems like an interesting approach for BOPlish, the different solutions have to be carefully evaluated and compared to ascertain if they cope with the unique constraints imposed by the BOPlish environment.

We begin this section by laying out prominent multicast techniques and related work in Sec. 6.1. We thereby take a look at different ALM system and characterize them to decide on a fitting approach for BOPlish. Sec. 6.2 introduces a concept that aims at extending BOPlish with pluggable group communication mechanisms. We continue by implementing a fitting ALM approach in Sec. 6.3 to validate the concept and finally evaluate the extended system in Sec. 6.4.

6.1 Background and Related Work

The Internet is built on top of a unicast, host-to-host communication model. However, todays services exploit the flexibility of the Internet that easily adapts to new use cases. One of these use cases is communication in groups. A lot of research has been done on group communication. This section gives an introduction to the research field and a short historical outline of the different technologies that exist today. We start with a general overview and classic IP Multicast approaches before digging into selected ALM protocols which operate on overlay networks. At last, we introduce the Publish/Subscribe paradigm which forms the basis for our extended protocol API.

Communication within a group can be done in multiple ways. A sender could send messages to each receiver (unicast) or flood the whole network with the information (broadcast). Both approaches are problematic. A unicast-like group communication (see Fig. 6.1a) replicates content in the network for every receiver. Senders and the network therefore have to cope with increased load. With broadcast (see Fig. 6.1b), the sender has to send content only once. This, though, imposes a high load on the network as the content is transmitted over every link at least once to reach each and every node in the network (duplicates can occur if two entities send the content to each other at the same time). This approach is highly inefficient if only a small group of receivers is interested in the content.



(a) Unicast (b) Broadcast

Figure 6.1: Unicast and broadcast network communication as occurring in IP networks

Multicast aims at providing a solution for efficient group communication. Receivers explicitly join a multicast group while they do not need knowledge about the actual location of the content but only about a group identifier. In a one-to-many communication scenario, this is

achieved by constructing a distribution tree with the sender as its root. When multiple senders are involved, a rendezvous point can be chosen that acts as the root of the distribution graph. Every recipient of the multicast group eventually receives the content during the traversal (see Fig. 6.2). Content dissemination in multicast is based on the push-paradigm. Thus, receivers do not query for new content. Instead, it is delivered automatically as long as they do not leave the multicast group.

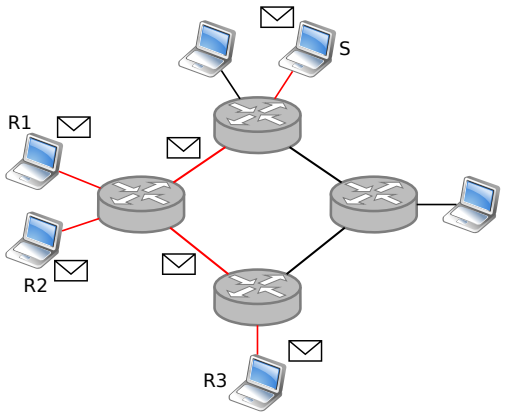


Figure 6.2: Multicast network communication as occurring in IP networks

6.1.1 IP Multicast

Multicast implemented on the IP layer promises efficient one-to-many and many-to-many communication while hiding the complexity of distribution graph generation and membership management in the network itself. The traditional addressing scheme in IP is the IP address. For multicast purposes, specific addresses have been reserved for IPv4[20] and IPv6[47]. The semantics for multicast IP addresses change such that specific addresses map to a group of receivers (a host group, as defined in the host-group model [25]) and not a single host.

Despite being around for a decade, IP Multicast suffers from deployment issues in the public Internet. BOPlish operates on the Internet. Thus, using IP Multicast is not feasible. Nevertheless, many ALM techniques are built on top of the ideas of IP Multicast. As such, it is an important building block for our ideas of group communication in BOPlish. IP Multicast functionality can be split into two domains – group and topology management.

Group Management

Group management in IPv4 Multicast is handled by Internet Group Management Protocol (IGMP) which was initially introduced in 1989 and allowed a client to negotiate a group join

with a neighboring IP Multicast-enabled router (which has to support IGMP, too). IGMPv1 [26] adds a method to the client's IP stack, allowing applications to de/register the participation interests in multicast groups:

```
JoinHostGroup  ( group-address, interface )
```

IGMP-enabled routers send periodic `Host Membership Queries` to discover hosts that have outstanding participation requests. The hosts then respond with a `Host Membership Report` that denotes the multicast group name (the multicast IP address). Hosts can also choose to send the `Host Membership Report` without a previous query in order for a timely join without having to wait for the periodic request. There is no explicit way for a receiver to unsubscribe from a group in IGMPv1. Instead, it has to wait for the router to timeout the subscription. Therefore, explicit leave-functionality has been added in IGMPv2:

```
LeaveHostGroup  ( group-address, interface )
```

RFC966 (1985)	First steps in extending DARPA with a multicast service; initial proposal of IGMPv1
RFC1112 (1989)	IGMPv1
RFC2236 (1997)	IGMPv2
RFC2710 (1999)	MLDv1 (IPv6 equivalent to IGMPv2)
RFC3376 (2002)	IGMPv3
RFC3810 (2004)	MLDv2 (IPv6 equivalent to IGMPv3)
RFC4604 (2006)	SSM extensions for IGMPv3 and MLDv2

Table 6.1: Important IP Multicast Group Management RFCs

Two multicast models exist: Any Source Multicast (ASM) and Source Specific Multicast (SSM). In ASM, any member of the multicast group can send content. As such, the network has to conduct a source discovery. In contrast, receivers in SSM groups are explicitly denoted by the group members up front. This, though, also assumes that receivers know the source address(es) in advance through another protocol or an external channel.

With IGMP v1 and v2, it is possible for clients to express the wish to participate in a multicast group denoted only by the multicast IP address. This implies that the network discovers all

multicast sources and routes these to registered recipient (ASM). As more sources are sending to a group the source discovery increasingly burdens the network. IGMPv3 adds the possibility to explicitly specify a list of senders to accommodate for SSM.

IGMP only operates on IPv4 networks. In IPv6, the group management protocol is called Multicast Listener Discovery (MLD). MLD exists in two versions, with the first one corresponding to IGMPv2 and the MLDv2 with IGMPv3. In contrast to IGMP, MLD is built into ICMPv6 which allows for a wide deployment of the group management protocol when the days of IPv6 come.

Topology Management

IPv4 Multicast uses IGMP to signal multicast interests between end nodes and routers. A routing protocol is needed to construct an efficient distribution graph consisting of the multicast-enabled routers, as well as the sources and receivers. Several techniques have been around for decades (e.g., DVMRP [102], MOSPF [61]) with new ones consistently surfacing (e.g., PIM-SM [29], BIDIR-PIM [42]).

Tab. 6.2 gives an overview about important milestones in IP Multicast routing protocols. These can be differentiated by taking a look at the distribution tree creation which can result in shared trees or source-based trees. With source-based trees (e.g., DVMRP, PIM-DM, MOSPF), a specific tree is constructed for every sender (denoted as (S, G)). The trees can be balanced and flat, allowing for an even load distribution. On the other hand, this requires a per-source state in every participating router. The costly construction and maintenance of source-based trees is the main argument for shared trees (e.g., PIM-SM, BIDIR-PIM) which root at a rendezvous point (denoted as $(*, G)$). A per-group state suffices for such trees which dramatically reduces state complexity for large groups. On the other hand, the trees are not optimal for all sender/receiver combinations.

Source-based tree approaches like PIM-DM assume that multicast group members are densely populated in the network. This is important as the network is flooded with state information for each (S, G) tuple and routers have to explicitly opt-out of the multicast participation (*push* model). Such algorithms lack efficiency when the network is not densely populated with participants. In the context of the global Internet, densely populated groups are unrealistic and a *pull* model (as used in PIM-SM) seems to be a better fit. In PIM-SM, every receiver joins a shared tree originating at a rendezvous point (RP) by sending a $(*, G)$-join. Senders can now send all traffic to the RP which sends the content down the tree. To improve efficiency, the RP can send a source-specific join back to the sender ((S, G)-join) which results in a shortest path from the source to the RP (thus, a concatenated tree is constructed). The sender can

RFC1075 (1988)	DVMRP - Source-based tree, Flooding/Pruning, based on RIP
RFC1584 (1994)	MOSPF - Source-based tree, Link State Flooding, based on OSPF
RFC2117 (1997)	PIM-SM - Shared tree, Rendezvous Point, protocol-independent
RFC3973 (2005)	PIM-DM - Source-based tree, Flooding/Pruning, protocol-independent
RFC4607 (2006)	PIM-SSM - Source-specific shortest path tree, explicit senders, protocol-independent
RFC5015 (2007)	BIDIR-PIM - Shared bidirectional tree, virtual Rendezvous Point, protocol-independent

Table 6.2: Important IP Multicast Routing RFCs and their characteristics (note that some of the RFCs have been obsoleted)

now use native multicast instead of the tunnel to the RP. In this state, all content originates at the RP. To further optimize the tree structure, the end routers can choose to send further (S, G)-joins directed to the original sender. This eventually shortcuts the shared distribution tree and results in the optimal tree structure – the source-specific shortest path tree (SPT) at the cost of increased state complexity.

More recent protocols (e.g., PIM-SSM) aim at directly building SPTs. PIM-SSM is a trimmed down version of PIM-SM and does not allow for $(*, G)$ joins at all. Instead of joining at an RP, routers send an (S, G) join directly to the source (similar to the PIM-SM shortcutting mechanism). SSM approaches are optimal in regards to the path distance (resulting in a Shortest Path Tree (SPT)) but only allow use cases where the sender is known up front (such as a radio station multicast) and does not change. As a tree is built explicitly for every sender, it does not scale well with many or moving senders.

PIM-SSM allows for static, small number of senders. In contrast, BIDIR-PIM tries to adapt the distribution tree to a large number of senders and receivers. It builds a bidirectional shared tree with a virtual RP. All senders route their packets up the tree using RPF instead of tunneling them to the RP via unicast. BIDIR-PIM therefore relieves the RP from coping with the tunneled traffic as well as the network from the (S, G) states. On the other hand, the explicitly constructed distribution tree is not optimal and no SPT can be built afterwards (like

in PIM-SM shortcutting).

6.1.2 Application-layer Multicast

According to the end-to-end design principles [81], functionality should be implemented in higher layers of the system stack unless a large performance benefit can be achieved that outweighs the cost of additional complexity in the lower layer. IP Multicast has been implemented in a lower layer due to the performance argument. In contrast to this and as an response to the deployment issues of IP Multicast, ALM approaches have surfaced which shift mulicast functionality to the application layer.

Instead of relying on the network, ALM approaches aim at embedding all the needed functionality in the end systems by spanning an overlay network between end nodes. Construction of the distribution tree lies in the hand of the application and is transparent to the underlying network which only has to expose elementary unicast forwarding capabilities. ALM avoids key shortcomings of IP Multicast: wide deployment is possible as no network support is required. This, though, comes at the price of lower efficiency due to the overlay network not having direct knowledge about the underlying topology. Chu et al. [19] have shown that the performance penalty introduced by overlay mechanisms that ALM system employ is low in the case of small and medium sized groups.

In contrast to IP Multicast, ALM solution are typically application-specific. Application designers run into trade-off decision which make it unfeasible to build a generic multicast layer that covers all use cases. Instead of one generic solution for many use cases, every application fosters a group communication layer that adapts to the use case. Typically, a custom group management protocol is employed instead of using a predefined standard like IGMP or MLD. We broadly separate ALM use cases into generic categories and outline their characteristics (Tab. 6.3).

Usage	**Group size**	**Number of senders**	**Amount of data**
Audio/Video streaming	high	low	high
Audio/Video conferencing	low	medium	high
Generic data transfer	medium	high	low-high
Event distribution	high	high	low

Table 6.3: Different usages require different group communication approaches

We introduced unstructured and structured overlays in Sec. 2. ALM systems operate on such overlay networks that do not rely on any support from the underlying network besides basic

unicast forwarding capabilities. Instead of using IP routing to communicate between peers, data forwarding is abstracted to the virtual links provided by the overlay [86, pp. 157–170]. ALMI [64] and ESM [19] are built on unstructured overlay topologies. Other approaches make use of structured overlays by implementing source-specific network flooding (CAN multicast [69]) or use a shared tree approach (Scribe [80]). We now continue to elaborate on the different ALM systems.

ALM on Unstructured Overlays

ALMI [64] builds on top of an unstructured overlay (a mesh). It is characterized as a mesh-first approach as the participating nodes establish a partially connected graph structure first and then build the distribution tree on top of the mesh. ALMI uses a centralized session controller that handles group management and tree creation/maintenance. Every peer is connected to the session controller over a point-to-point unicast link. The session controller computes a minimal spanning tree between all participating peers using a performance metric (e.g., delay). Nodes continually probe other nodes using the metric and send results to the controller which uses the information to improve the tree structure. This gets increasingly costly for large groups ($\mathcal{O}(N^2)$). Therefore, ALMI limits the dissemination of the probe messages leading to a sub-optimal graph. The constructed tree is a bidirectional shared tree which is sub-optimal first but enhanced over time with the session controller evaluating the performance measurements [86, pp. 157–170].

Due to the use of a centralized session controller, implementation is expected to be rather straight-forward and a high degree of control over the tree structure is granted. The bidirectional tree allows for efficient any-source multicast. On the other hand, ALMI suffers from the scalability issues that occur in centralized system due to the session controller. Moreover, it represents a single point of failure. Contrary to ALMI, ESM [19] does not rely on any central component but is built solely from participating end hosts.

Group management and data replication in ESM is handled by the Narada protocol. Narada employs a mesh-first approach by organizing nodes in a partially connected mesh structure. Every group member keeps track of all other participants. Nodes joining the system announce themselves via a flooding mechanism that broadcasts the join to all peers. Periodic heartbeats are then flooded to detect failing/leaving peers via a maintenance mechanism. Scalability of this approach is obviously limited. ESM is explicitly designed only for small to medium sized groups that can cope with the high overhead of maintenance traffic (the original paper [19] specifically targets data center applications). After the mesh has been constructed, source-specific multicast trees are built on top for every publishing entity using RPF (comparable to

PIM-SSM) [86, pp. 157–170].

ALM on Structured Overlays

Various ALM approaches based on structured overlays have been developed, the most prominent being CAN mulicast [69], Bayeux [108], Scribe [80] and SplitStream [15]. Two general approaches exist, either maintain a group-specific sub-overlay (CAN) and flood information or build distribution trees from all members. A more recent approach is BIDIR-SAM [101] which also constructs distribution trees.

CAN multicast is a flooding-based approach where participants of a group build a per-group "mini"-CAN. A group is identified by an identifier G concatenated with the user identifier of the creator. The node that is responsible for the hash value of the group identifier acts as a bootstrap node for a newly created mini-CAN. Content dissemination then happens in a broadcast-manor along the mini-CAN. Due to the CAN mechanics, duplicates can not be ruled out if the coordinate space is split unfavorable by the algorithm [86, pp. 157–170].

Scribe is a Publish/Subscribe (Pub/Sub) system that builds on the Pastry DHT [16], a structured overlay system similar to Chord. By using a DHT substrate, Scribe inherits the scalability, self-organization and fault-tolerance properties. Subscribers can join groups and senders can publish events to that group (topic-based Pub/Sub). A group is identified by an identifier $topic$ which is hashed to a key ($topicId$). Scribe uses Pastry to store the $topidId$ in the DHT at the numerical closest peer to that key. This peer is then used as a central rendezvous point and the root of the distribution tree for the specified $topic$. Distribution happens according to a per-group shared tree that is built using reverse path forwarding. Scribe provides weak reliability guarantees but can be extended to support reliable and ordered delivery [80].

The authors of the SplitStream [15] approach argue that single-tree-based multicast systems (like Scribe) are not well-suited for highly cooperative environments. This is due to a relatively small number of interior nodes of the tree carrying large responsibility for the rest of the tree. An interior node with limited bandwidth capabilities thus affects all children nodes. The authors propose a forest that is composed of multiple interior-node-disjoint multicast trees. Content is split into k stripes and each stripe is distributed over an disjoint tree. Forwarding load is thereby distributed among all participating peers. By overcoming the unbalanced forwarding load, high-bandwidth applications like video distribution are possible. Moreover, a tree dropout only affects a single stripe. Applications can choose to include redundancy information in the other stripes to mask the effect of drop outs. SplitStream relies on Scribe and uses a separate Scribe tree for each of the k stripes.

Bayeux [108] is an approach to source-specific ALM on structured overlays. It uses prefix-

routing-based Tapestry [107] as its DHT substrate and creates shortest-path distribution trees centered around sources. In contrast to Scribe, distribution trees generation is not receiver-initiated. Instead, forwarding states are set up from the source towards each receiver to create a shortest path tree for each client subscription.

BIDIR-SAM [101] operates on the prefix structure of an overlays key space. The authors argue that the major deficit of tree-based approaches lies in the limited reliability and increased vulnerability in case of node or link failures. This weakness is overcome by creating virtual distribution trees with vertices mapping to a prefix (thus, a group of nodes). The prefix is variably bound to nodes in the system and no designated rendezvous point is required. The resulting distribution tree is a bi-directional, shared tree and data is forwarded according to source-specific shortest paths. Such a tree structure is particularly well suited for any source multicast.

6.1.3 Common APIs for Group Communication

We presented a multitude of ALM approaches that typically use APIs specific to the technology (e.g., the DHT underlay) in use. Comparing and evaluating different approaches is tedious because the application has to adapt to each API. Therefore, the research community expends effort to find a common API that hides the technology-specific APIs from the application [67, 99].

Many of the introduced approaches to overlay multicast are based on the Publish/Subscribe paradigm. Entities in a system based on Pub/Sub are publishers, subscribers and an abstract event notification service [30]. Pub/Sub systems can be broadly classified into into topic-based or content-based systems based on their subscription model. Subscribers explicitly issue an interest in a specific type of content or an explicit topic identifier. Information related to the content or topic is then distributed from publishers to subscribers using the event notification system. A decoupling of publishers and subscribers is achieved by leaving them ignorant of one another (a decoupling in space). Thus, publishers do not have any knowledge about the registered subscribers. Subscribers, on the other hand, do not need to know where the content originates. Instead, only a group name is required that is not tied to the location of the content. Additionally, Pub/Sub systems might also be decoupled in time which means that a subscriber can be disconnected during the publishing of a message. The message is delivered when the subscriber reconnects. Conversely, the publisher might already be disconnected while the subscriber still receives the message [30] at a later time. Applications built on top of the paradigm are naturally loosely coupled due to the introduced decoupling.

Pietzuch et. al [67] propose a light-weight Pub/Sub API with three goals: ease-of-use,

interoperability and extensibility. The authors grade different Pub/Sub approaches into compliance levels. The first compliance level (L1) resembles an abstract Pub/Sub interface that does not dictate implementation details in any way. The L1 API is accompanied by optional calls that eventually re-introduce a binding to technology, e.g., `renew_lease()`. This is not preventable as some Pub/Sub systems rely on, e.g., periodic re-subscription messages while others do not. The proposed L2 compliance shares the same over-the-wire protocol based on XML-RPC, specifying error reporting message types and authentication using an HTTP mechanism. At last, the L3 API introduces a common model for event data and subscriptions. The model is based on XML and allows multiple systems to use a common data structure. As such, events and subscriptions can be formatted in the same way, independent of the Pub/Sub system in use.

Waehlisch et. al [99] present an approach for transparent, hybrid multicast communication. The included, application-facing API is not bound to Pub/Sub systems as in [67] but suitable for any multicast-based system. The authors argue that application development should be decoupled from the multicast flavor (such as ASM and SSM) as well as the layer it operates on (e.g., IP or application layer). Further, the approach allows inter-technology transmission transparent to the multicast flavor in use via Inter-Domain Multicast Gateways that connect the different multicast islands.

To achieve transparent communication spanning multicast technologies, a custom naming scheme based on URIs is used. Groups are identified by an identifier that is unique within a namespace. The namespace describes the syntax of the group which can be adapted to the environment (e.g., a generic IP address or a SIP conference URI). The authors describe the ability to add future namespaces using a public registry which allows easy extendability for new applications.

A middleware hides the technology-specific APIs of the multicast technologies from the application and in turn offers an application-facing API. This API is divided into four parts that handle group management, data distribution and optional extensions for socket options and service calls. The API is centered around the URI scheme. URIs are used to, e.g., join/leave groups or send/receive data.

6.2 A Generic ALM Layer on BOPlish

Today's ALM systems are widespread but hidden in the application and proprietary (e.g., Spotify [55]). In contrast, BOPlish enables application developers to build provider-independent applications using a common URI scheme. We aim for the same ambitions regarding the group communication layer. In BOPlish, the targeted application domain is manifold. We foresee applications ranging from typical source specific use cases such as central video distribution to any source use cases with large proportions of sending entities such as social networks or group chats.

A multicast solution has to be adopted to the actual application domain. Approaches to ALM are trade-off decisions, making it unfeasible to build a single generic multicast layer on top of BOPlish due to the manifold use cases. Instead of covering all possible use cases or restricting the architecture to a single generic multicast solution, we propose additions to the BOPlish core that allow pluggable multicast systems. This includes additional group naming semantics to the BOPlish URI scheme and a generic interface that is made available to the protocol facility. Apart from the changes to the application core, we exemplify the implementation of an ALM system. The implementation shows the general feasibility of a multicast layer on top of BOPlish and verifies the concept. This implementation is not supposed to fit all use cases. Rather, other ALM implementation should be added in the future to accommodate to the specific application requirements.

6.2.1 Generic Group Communication API

We introduced two approaches to common APIs for group communication [67, 99]. Using a generic API that can adapt to multiple group communication technologies is vital when requiring interchangeable ALM systems. The API proposal by Waehlisch et. al [99] includes detailed guidance and not only allows interchangeability of different multicast implementations but also a way to interconnect islands of different multicast technologies (e.g., IP Multicast). This is a major benefit but also requires compatible Inter-Domain Multicast Gateways which are currently not available (and not feasible until stable, server-side WebRTC implementations emerge). Adapting BOPlish to the common API is certainly possible. A generic adapter to Web-based applications would allow interesting use cases and connect the Web environment seamlessly. We still decided not to use the API for our prototypical solution. Our goal of the group communication layer is to experiment with the general feasibility of ALM over WebRTC-based systems. For these reasons, we considered a simpler API approach sufficient for our current usages.

The API proposed in [67] is divided into three compliance levels. Each level increases the amount of dictated properties. For Level 1 compliance, the interfaces pictured in Lst. 6.1 are required. Level 2 and 3 dictate an XML-RPC API and the data format used on-wire. Level 2 and 3 are again not considered for our approach due to the same reasons stated above. The Level 1 API is very generic and intuitively to use but does not open access to systems other than Pub/Sub. Moreover, the benefit of connecting different multicast islands is not considered in the API proposal.

```
1 subscribe(filter_expr, notify_cb, expiry) -> sub_handle
2 unsubscribe(sub_handle)
3 publish(event)
4 ~notify_cb(sub_handle, event)
```

Listing 6.1: Main interfaces of the Level 1 Pub/Sub API proposal [67]

Subscribe is called with a filter expression chosen by the implementation and a notify callback that is called whenever a message matching the expression arrives. A subscription handle is returned that allows the application to refer to this subscription at a later time. An expiry time can be specified for the subscription while setting it to zero indicates infinite leases. The filter expression is bound to the implementation in use. In a topic-based Pub/Sub system it may consist of a topic name while content-based Pub/Sub specify the attribute specification required for a subscription match. The unsubscribe call removes a subscription indicated by the handle argument.

The publish call takes a single argument: the event to be published. Events are generic in the sense that the implementation dictates the actual properties and data format. At last, the notify callback is invoked each time a matching event arrives. Apart from the event data, the matching subscription handle is included. This allows the client to determine the subscription that caused the notification.

The described API complies to the Level 1 API proposed in [67]. Something that is not handled here is state indication to the client. We therefore decided to decorate the API with promises that are returned by each call, indicating success and error messages in an asynchronous way. Promises either resolve or reject according to their state and call the according callback once the state is set. The usage of promises is completely optional in our API. This allows all calls to be non-blocking as usual in JavaScript.

```
1 var p = publish({...});
2 p.then(function() {
3     console.log('publish successful!');
4 }, function(err) {
```

```
5       console.log('publish_failed!');
6 });
```

6.2.2 Group Naming

After specifying the user-facing API, we lie the focus on the naming of groups. To subscribe or publish in a group, end hosts use distinct group names or expressions indicating a type of content. In IP Multicast, a specific range of IP addresses is used to identify such groups. Receivers express their interest by using IGMP to relay the interest to a neighboring multicast-enabled router. Such an interest can be expressed using ASM or SSM semantics. In ASM, only the multicast IP address of the group is specified (denoted as $(*, G)$). In SSM, on the other hand, the identifier (S, G), apart from the group name, explicitly specifies the source(s).

In ALM scenarios, naming is more flexible as it does not depend on IP addresses. As such, the different approaches each incorporate their own naming scheme. Scribe, for example, uses source-independent identifier built from the group name concatenated with the name of the creator. This identifier is then hashed using a collision resistant hash function that ensures a uniform distribution, allowing it to be spread uniformly across the key space and therefore the nodes in the system (using the DHT).

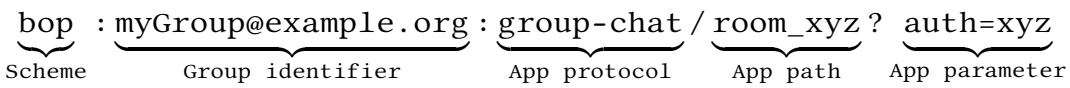
From the very beginning of our work, we designed the user-centric identifiers used by BOPlish to be extendable for group communication claims. To characterize the group naming approach used in BOPlish, we recapitulate the BOPlish URI one-to-one scheme previously described in Sec. 3.3.2:

$$\texttt{bop:}\underbrace{\texttt{alice@example.org}}_{\text{User identifier}}\texttt{:}\underbrace{\texttt{one-to-one-chat}}_{\text{App. protocol}}\texttt{/}\underbrace{\texttt{room\_xyz}}_{\text{App. path}}\texttt{?}\ \underbrace{\texttt{auth=xyz}}_{\text{App. parameter}}$$

The BOPlish user identifier is bound to an identity and consists of a user name accompanied by an identity provider (e.g., `alice@id.com`). In a group communication scenario, we do not want to bind user identities to the group name as this would prevent groups from being logically bound to multiple users. We opted for names that keep the BOPlish semantics intact by exchanging the user identifier with a group identifier. This approach leaves users and groups completely ignorant of each other. The group identifier resembles the group name accompanied by an identity provider and is hashed using a collision-resistant hash function identical to the user identifier in BOPlish one-to-one (see Sec. 3.3.2). The hash is then stored in

the BOPlish name resolution system, indicating the existence of this group.

Only the group identifier of the URI is used to identify the group (e.g., `myGroup@example.org`). This implies that protocol semantics of the URI stay intact. Multiple protocols can operate in the same group (e.g., `group-chat` and `file-share`). As an example, a group chat protocol instance exploiting the multicast facility could use an identifier like this:



The group name is string-based and can be freely chosen by the protocol. The group identifier generated from the group name and identity provider is composed into a BOPlish URI, also containing the protocol type and parameters. Thus URI is fed into publish calls along with the message to be published. Upon subscribing, a URI is fed into the subscribe call along with a notification callback and the expiry time. The call returns a handle that can later be used to unsubscribe.

```
1 publish({
2       uri: <<bopuri>>,
3       payload: <<payload>>
4 });
5 var callback = function(err, msg) {
6       console.log(msg);
7 };
8 var handle = subscribe(uri, callback, 0);
9 unsubscribe(handle);
```

6.3 Scribe ALM on BOPlish

We now continue to lay out an ALM approach that is built on top of our concept and implement it afterwards. We introduced many ALM systems and pointed out that choosing a single fitting ALM approach to all use cases is unfeasible. The approaches can be broadly separated into flooding-based (e.g., CAN multicast) and tree-based (e.g., Scribe, BIDIR-SAM). Castro et. al [17] have shown that tree-based ALM approaches generally outperform flooding-based ones when operating on a P2P overlay. This still leaves us with a multitude of different approaches, each with its own pros and cons. It has been pointed out by [10] that conventional tree-based structures like the ones employed by Scribe and SplitStream are not particularly well suited for

the unreliable characteristics of cooperative distributed environments. BIDIR-SAM mitigates these problems by creating virtual distribution trees based on prefixes. BIDIR-SAM requires a DHT overlay that forwards according to prefixes. BOPlish currently uses Chord as its DHT overlay. This would need to be replaced by, e.g., Pastry. We decided to choose Scribe [16] as our first approach to an ALM system in BOPlish despite the criticism that was hold up against it due to the following reasons. Scribe is relatively easy to implement and has proven its scalability in large-scale simulations ([17]) and implementations [28]. The general characteristics should help deciding if an ALM approach on top of WebRTC is feasible. Moreover, implementing SplitStream (which is built on Scribe) afterwards is considered relatively easy.

Scribe builds on Pastry [79], a DHT similar to Chord that also uses a divide-and-conquer approach in a ring-based key space [17]. The difference lies in the routing policies. Chord routes clockwise in the ring-based key space while Pastry uses prefix trees. Our solution is based on Chord, not Pastry, as Chord is already implemented and used in the name resolution mechanism of BOPlish (see Sec. 4.5). DHTs are easily interchangeable later due to the key-based routing API that is offered by the BOPlish Router component (see Sec. 4.4.4).

BOPscribe inherits its name from the BOPlish environment it runs on and the related Scribe ALM approach. Fig. 6.3 shows a simplified example of a group communication scenario using Scribe. First up, Alice creates a topic using a group identifier. This identifier is hashed into a key and stored in the DHT. In this case, the peer responsible for the key is Carol. From now on, other peers can subscribe to the group using the same identifier. This subscription is propagated using the DHT overlay until the root is reached. Whenever a peer wants to publish data to the group, it sends the data to Carol who distributes it to the registered subscribers.

Scribe offers an API pictured in Fig. 6.4. The Scribe-specific API needs to be wired to the generic group communication API and naming semantics described in Sec. 6.2. Lst. 6.2 shows (in simplified terms) how this is done. The create call of Scribe is mapped to a publish call with the data property set to null. Apart from the simplified calls described in Lst. 6.2, promises are returned that signal success and error states to the protocol. Moreover, the subscribe call passes in an expiry time that is used for the periodic tree maintenance mechanism described above.

```
1 GAPI.publish({ uri: <<uri>>, data: null });
2 // Scribe.create(uri.credentials, uri.groupIdentifier);
3
4 GAPI.publish({ uri: <<uri>>, data: <<data> });
5 // Scribe.publish(uri.credentials, uri.groupIdentifier, data);
6
7 var handle = GAPI.subscribe(uri, notify_callback, expiry);
```

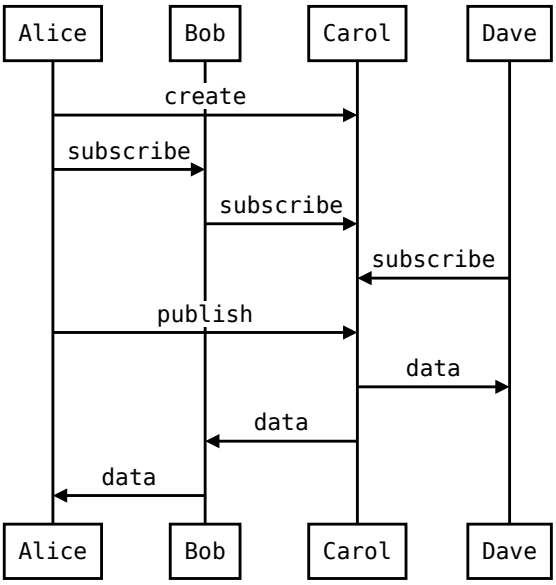


Figure 6.3: Scribe ALM sequence diagram with Alice creating a group at the rendezvous point Carol. After creating the group, other peers can subscribe to the group identifier and thereby create the distribution tree using reverse path forwarding. All data send to carol is recursively propagated down the distribution tree.

create(credentials, topicId) creates a topic with topicId. Throughout, the credentials are used for access control.

subscribe(credentials, topicId, eventHandler) causes the local node to subscribe to the topic with topicId. All subsequently received events for that topic are passed to the specified event handler.

unsubscribe(credentials, topicId) causes the local node to unsubscribe from the topic with topicId.

publish(credentials, topicId, event) causes the event to be published in the topic with topicId.

Figure 6.4: Topic-based Pub/Sub API used by Scribe [16]

```
8 // Scribe.subscribe(uri.credentials, uri.groupIdentifier,
     callback) -> uri
9 // callback calls notify_callback with (uri, data)
10
11 GAPI.unsubscribe(handle);
12 // Scribe.unsubscribe(uri.credentials, uri.groupIdentifier);
```

Listing 6.2: The Scribe API (Scribe) can easily be wired to the BOPlish Group Communication API (GAPI)

An explicit `onmessage` callback has to be set by the protocol to receive messages this peer receives due to his subscriptions. The `credentials` parameter can be used to validate requests at the rendezvous point or on intermediate hosts. In our prototypical implementation, we do not use credentials and therefore push further security-related analysis to future work.

Arising question regarding security are: who determines if the peer is authorized for the specific action and how to authenticate the identity? An interesting approach for the latter question is to use the `topicId` itself to authenticate at an identity provider similar to BOPlish one-to-one.

6.3.1 Implementation

After laying out the concepts for this approach we now continue to implement BOPscribe. We identified pluggable ALM schemes to be of high value as a single generic multicast layer that can cope with all use cases is deemed well-nigh impossible. An important goal of the implementation efforts is thus a loose coupling of the group communication implementation and the core BOPlish library. This allows to exchange the ALM system in case the use cases fits another approach.

We dig into the implementation details of BOPscribe by first giving an architectural overview before exposing the changes made to the BOPlish core. Further, we give API examples of protocols leveraging the group communication layer to give the reader a feel for the API. Fig. 6.5 shows the extended architecture of BOPlish. BOPscribe interfaces the Router component (see Sec. 4.4.4) and is called by the BOPClient whenever an application utilizes the group communication API (see Sec. 6.2). Received messages get passed up to the BOPclient which in turn passes them to the respective BOPlish application protocol.

6.3.2 DHT Underlay

The BOPlish Router component offers a KBR-like API [22] and uses Chord [90] as its underlying DHT mechanism. Our ALM approach is similar to Scribe [16] which is originally based on

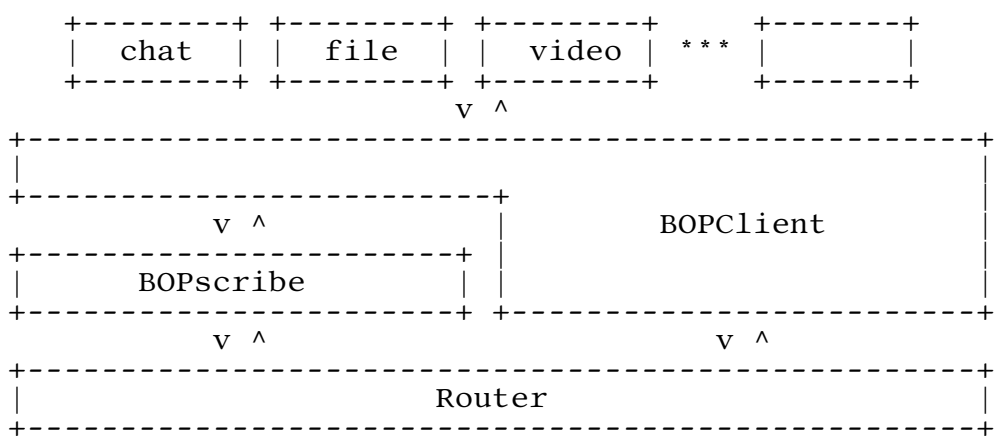


Figure 6.5: How BOPscribe fits into the BOPlish architecture

Pastry instead of Chord. Both DHTs maintain a similar ring-based topology but Pastry routes on trees built from neighboring peers while Chord maintains a finger table with logarithmically increasing distance between the peers.

The underlying DHT could easily be exchanged due to the standardized KBR API. As of now, we stick with Chord but it would be interesting to compare BOPscribe when using different underlying DHT mechanisms. The DHT offers the following interfaces: `put()`, `get()` and `remove()`. The KBR API abstracts the DHT interfaces to a single `route()` call, accompanied by `forward()` and `deliver()` [22]. To route messages between peers, `route()` is called with a key and the message as its parameter. The `forward()` call is invoked at every peer when a message arrives that is to be forwarded and gives applications the possibility to modify the passing message. This is crucial for our Scribe-based approach which relies on consuming messages that are en route to the rendezvous point and not actually addressed to the peer that consumes the message. At last, the `deliver()` call gets invoked when someone routes a message with a key the peer is responsible for.

In BOPlish, the KBR API has been adapted to accommodate to the asynchronous JavaScript environment. Message interception capabilities required by `forward()` calls in BOPlish are implemented as follows. The Router offers a `registerInterceptor()` which takes a callback function as argument (see Lst. 6.3). This callback gets invoked each time a message passes the peer. Parameters are the passed message and another callback used to propagate the message to the next interceptor or, when all interceptors are processed, to the routing layer. The first parameter of the propagation callback is an error message that cancels propagation and signals the error message to the sender once set. The other parameters are the message

that can be modified in place and a boolean that drops the message when set. When drop is set, successful delivery of the message is signaled to the original sender.

```
boprouter.registerInterceptor(interceptor.bind(this));
function interceptor(msg, next) {
      console.log('intercepted message', msg);
      var error = null;
      msg.someParameter = true;
      var drop = false;
      next(error, msg, drop);
}
```

Listing 6.3: Register message interceptor

Apart from the additional route interception mechanism, no changes were made to the BOPlish core for the BOPscribe implementation. This leaves us confident that other ALM approaches based on a KBR-like API are also implementable without tinkering with the core functionality. We now fill the concept of BOPscribe with the implementation details. This includes specifying the message formats and the reaction of the system to each of the messages.

Group Creation

To create a group, BOPscribe crafts a CREATE message and routes it in the direction of the group key (`sha(myGroup@id.com)`). The peer responsible for this key receives the message and from now on acts as the rendezvous point for the specified group hash. The rendezvous point stores the group in its local key storage. This allows the Chord DHT to replicate the group entry to its successors using the maintenance mechanism. A such, when the rendezvous point fails, the successor will automatically take over.

```
var create = { // recipient sha({{Group identifier}})
      type: messageTypes.CREATE,
      createdOn: {{Date of creation}},
      credentials: {{Authorization credentials}}
};
```

In the same way, groups can also be removed by issuing a REMOVE message. It is routed through the DHT until the top of the tree is reached. The root then deletes the group and denies further subscriptions/publications.

```
var remove = { // recipient sha({{Group identifier}})
      type: messageTypes.REMOVE,
      groupId: {{Group identifier}}, // e.g., myGroup@id.com
```

```
4       credentials: {{Authorization credentials}}
5 };
```

Group Subscription

Every peer stores their own subscriptions in a list. As long as the peer wants to stay subscribed to the group, it issues periodic SUBSCRIBE messages containing the peer ID in direction of the group identifier hash. The next hop intercepts the message, consumes it and adds the peer to the list of registered subscribers for this group. All peers periodically craft SUBSCRIBE messages to subscribe themselves to all current subscriptions. Thus, this procedure recursively subscribes all peers en route until the root of the tree is reached. If no SUBSCRIBE message arrives for a specified amount of time, the subscription times out and is removed by simply deleting the entry from the subscription list which is again propagated if the subscription list is now empty.

```
1 var subscribe = { // recipient sha({{Group identifier}})
2       type: messageTypes.SUBSCRIBE,
3       peerId: {{Peer ID}},
4       credentials: {{Authorization credentials}}
5 };
```

Leaving the group can either happen by waiting for the timeout or by crafting an explicit LEAVE message, again containing the peer ID. When a peer receives a leave request, it checks if it has remaining subscribers and only propagates the leave if not.

```
1 var leave = { // recipient sha({{Group identifier}})
2       type: messageTypes.LEAVE,
3       peerId: {{Peer ID}},
4       credentials: {{Authorization credentials}}
5 };
```

Group Publication

Publication requires the sender to route messages to the root of the tree. The rendezvous point receives the message and consumes it. It could check if the publisher is authorized to publish but this is currently not implemented. Afterwards, DATA messages are created that hold the content of the publication. They are sent to all subscribers in the subscription list of the peer. Typically, this list is small as it only contains subscribers that are direct neighbors due to the recursive creation of the subscriber list.

```
var publish = { // recipient sha({{Group identifier}})
      type: messageTypes.PUBLISH,
      payload: {{Protocol payload}},
      credentials: {{Authorization credentials}}
};
```

Whenever a peer receives a DATA message, it maps the included group hash to its subscription list and routes the message to each known subscriber.

```
var data = { // recipients {{subscriptionList}} for groupKey
      type: messageTypes.DATA,
      groupKey: {{Hashed group identifier}},
      payload: {{Protocol payload}},
      credentials: {{Authorization credentials}}
};
```

We covered all message types of BOPscribe from creation of a group to the actual data distribution. It can be seen that the group identifier is never transmitted in clear text but only a hash value of it which is created locally. Thus, the group names are naturally private as long as the hash function is considered secure.

6.3.3 Leveraging BOPscribe

Protocols leveraging the group communication layer use the BOPlish group communication API introduced in Sec. 6.2 to communicate in groups. Every API call is mapped to the according functionality as described in Sec. 6.3. Lst. 6.4 shows a minimal example of such a protocol (i.e., create, subscribe, publish to a group).

```
// specify onmessage callback
var onmessage = function(uri, msg) {
  console.log('Topic:␣%s,␣Msg:␣%s', uri.groupIdentifier, msg);
};

var bopuri = new BopURI('bop:myGroup@id.com/my-protocol');

// create group myGroup@id.com
var p = proto.group.publish({uri: bopuri, data: null});

p.then(function(){
   // subscribe to 'myGroup@id.com'
   proto.group.subscribe(bopuri, onmessage, 1000);
```

```
14   // send textual data
15   proto.group.publish({
16     uri: bopuri,
17     data: "my_message"
18   });
19   // send json-encoded data
20   proto.group.publish({
21     uri: bopuri
22     data: {
23       message: 'my_message',
24       username: 'alice'
25     }
26   });
27 });
```

Listing 6.4: Minimal example of a protocol leveraging BOPscribe

Published data can be either string-based or encoded in JSON, similar to the `proto.send()` method introduced by BOPlish one-to-one. Currently, binary data such as images has to be encoded into a textual form via, e.g., Base64 to send it over BOPlish. The overhead that is introduced by encoding binary data can be reduced using JavaScript `ArrayBuffer`. If transmitted over WebRTC, the Data Channel has to be specifically set up for such binary transmission. We currently use the same Data Channel instance for data distribution and DHT/ALM-related maintenance traffic. As such, binary transmission is not supported. This decision is ruled based on the current status of WebRTC implementations that do not yet support ArrayBuffer to their full extent. As an example, the Chrome browser fails when sending messages bigger than $64\ kB$ and applications have to manually chunk the data[1]. We therefore opted to stick with a single Data Channel for now and wait for the implementations to stabilize before deciding on further steps.

6.4 Evaluation

We laid out the conceptional details and the implementation of our ALM approach. To conclude the section, we now continue to evaluate the system. We evaluated BOPlish one-to-one using the emulation environment described in Sec. 5. The same environment is used to measure scalability-related metrics in BOPscribe. As such, the same restrictions apply and we are currently limited to a LAN environment with high bandwidth and low delay links between

[1] `https://code.google.com/p/webrtc/issues/detail?id=2270`

peers. As a future addition, it would be valuable to test the implementation using larger numbers of geographically distant peers. This could, e.g., be achieved using a PlanetLab based[2] setup. This, though, should be delayed until the WebRTC specification is finalized and stable server-side WebRTC implementation emerge.

Multi-hop performance has already been evaluated in Sec. 5.2 and the results are directly applicable for BOPscribe. As such, the structure of the distribution tree that is recursively generated for each group is the main criteria for this evaluation. We generate tree traces by appending peer IDs to the DATA message flowing from the rendezvous point down to the members of the group. We obtain the complete structure by collecting the traces at the leafs and combining them to a tree.

An application has been created that acts as a testing front end to the BOPscribe layer (Fig. 6.6). Using this application, we validate protocol mechanisms and measure performance in the emulation environment.

The environment of this evaluation is the same that is specified in Sec. 5.2 with the exception of the DHT configuration. The original Chord paper suggests to use a SHA algorithm to hash peer and content identifiers to the same key space. When using SHA-1, the resulting key space lies in the interval $0...2^m - 1$ with $m = 160$. The Chord finger table maintains entries for every key at $n+2^{k-1}$ with $n = peerId$ and $k = 1....m$ (i.e., m entries). This leads to a problem in our implementation: Web browsers currently only support a limited number of concurrent Peer Connections. For our evaluation, we want to use smaller-sized finger tables that can be fully populated. To achieve this, we restrict the key space to $0...2^{16} - 1$ by calculating the modulo of the hash and the key space. This leads to fewer entries in the finger table at the price of increased collision probability (which is neglectable for our small test groups). The hash for an identifier is created as follows: $sha1('identifier') \bmod 2^{16}$. This eventually chops of 144 bit from the 160 bit result. We expect the statistical properties of the hash function to hold but a proof is considered complex. We decided to run a test to verify the uniform distribution of the resulting keys. The results graphs and source code of this test can be found on the CD that is included with this thesis (/evaluation/uniform-distribution*). It shows that the resulting limited key space is still uniformly distributed.

6.4.1 Scalability in Larger Groups

BOPscribe is supposed to operate in BOPlish User Communities. Such communities are not bound to a specific group size but are expected to be of sizes in the hundreds. Our emulation environment currently cannot handle hundreds of peers due to multiple issues: The underlying

[2] https://planet-lab.org

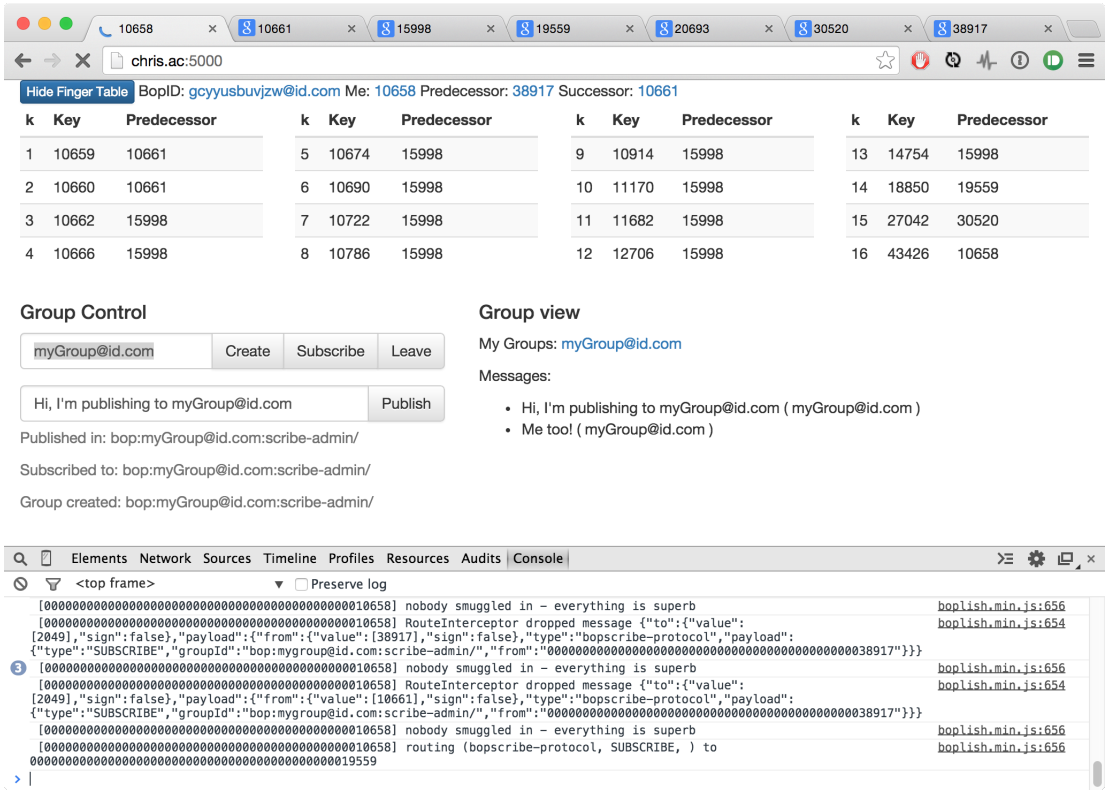


Figure 6.6: BOPscribe test application displaying the finger table of this peer and a GUI allowing to leverage the group communication API.

WebRTC library is not finalized and fragile in some parts[3]. Moreover, the libraries used in Google Chrome and Mozilla Firefox happen to generate high system load due to bugs, requiring vast amounts of computing resources[4,5].

In the emulated environment, many peers share few systems. If individual peers abruptly require high amounts of resources, they affect the other peers on the system. This can lead to abrupt detachment of all peers due to increased timeouts. In this case, our system breaks down as it cannot handle such a large amount of churn. We therefore choose to pick group sizes where the system could work entirely stable and derive scalability properties for larger groups from the results.

In this test all members of a User Community subscribe to a single topic. BOPscribe builds a distribution tree rooted at one of the peers. We measure average, as well as min/max hop

[3] https://github.com/js-platform/node-webrtc/issues/144
[4] https://bugzilla.mozilla.org/show_bug.cgi?id=861050
[5] https://bugzilla.mozilla.org/show_bug.cgi?id=979716

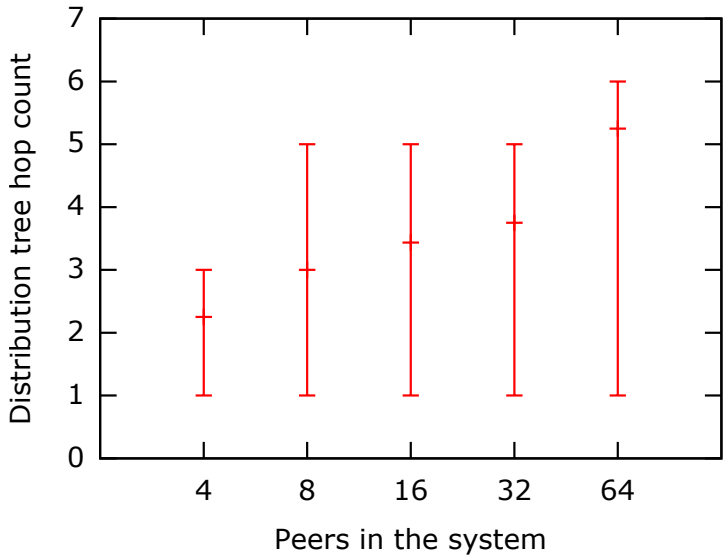


Figure 6.7: Average (with min/max markers) BOPscribe distribution tree length with increasing group size

counts in such trees while increasing group sizes (see Fig. 6.7). It can be observed that the average number of hops rises on a near-linear basis despite doubling the group size in each iteration of the test. The minimum hop count is always 1. This is obvious as the rendezvous point is also part of the group. The maximum hop count in our test stays at 5 hops for group sizes of $8-32$, marking the worst case for a leaf node. With 64 peers in the system, the average hop count increases to 5.25 which is roughly in line with the expected number of routing hops the Chord DHT dictates ($\frac{1}{2}\ log_2(N)$).

6.4.2 Distribution Tree Maintenance

A central aspect to the system is its behavior in case of failing or leaving peers. Web-based applications are faced with rather high rates of churn. As such, we evaluate the impact of failing rendezvous points as well as members of the distribution tree. BOPscribe constructs a per-group distribution tree rooted at the rendezvous point. To visualize such a tree, we join 17 peers that form a User Community. A group is created and a rendezvous point is thereby elected. The other 16 peers subscribe to the group and form the distribution tree. The initial distribution tree created by the 17 peers is pictured in Fig. 6.8.

We can observe that the tree is not very well balanced. The reason for this is that the structure is coupled to the DHT routing and therefore not influenceable. Moreover, messages

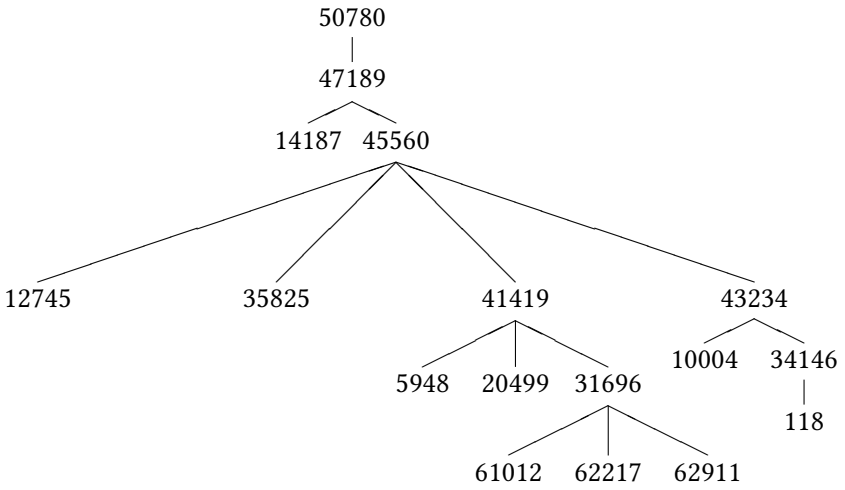


Figure 6.8: Initial BOPscribe distribution tree rooted at rendezvous point 50780

are always routed to the predecessor of the group hash (47189) before reaching the rendezvous point again caused by the Chord mechanics [90].

Rendezvous Point Failure

Data is always distributed from the root of the tree down to all children in the subscriber list. The root of the tree is a quasi-random rendezvous point. If the rendezvous point fails, the tree has to be reconstructed. We visualize the impact of such a failing root by closing the corresponding node with peer ID 50780 and redraw the tree after it has stabilized. The amount of time it takes to stabilize is coupled to the DHT maintenance timeouts as well as the periodic re-subscriptions issued by BOPscribe. In our LAN environment, the timeouts can be set to low values but have to be increased when operating on the public Internet. Fig. 6.9 shows the result of a failing root after the stabilization.

Peer 61012 is now the rendezvous point as it was the previous successor of 50780. It is therefore detached from its old parent node 31696. It can be seen that the departure of the rendezvous point has only affected a small number of peers (namely, the direct neighbors printed **bold**). Publishing to the group continues seamlessly, even though some messages are lost at the moment the original rendezvous point fails because the subscription list of the new rendezvous point has to populated (i.e., when peer 47189 issues the periodic re-subscribe message).

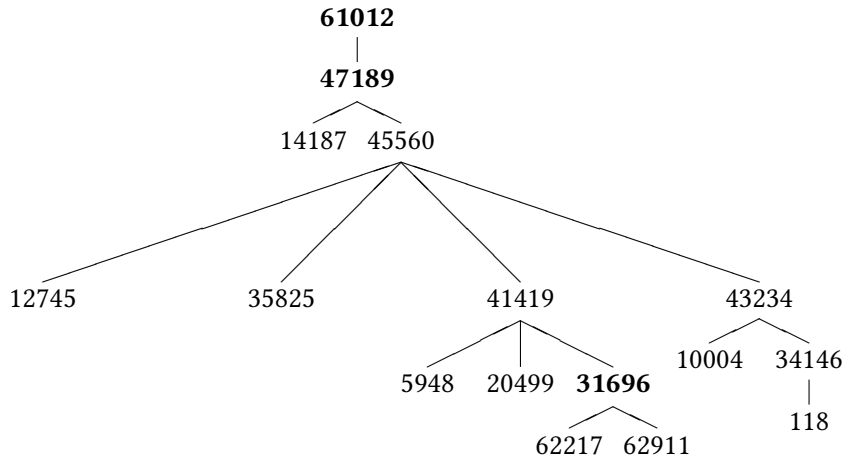


Figure 6.9: BOPscribe distribution tree after rendezvous point failure

Inner Failure

After investigating rendezvous point failures, we continue to take a look at inner failures, i.e., how the system reacts to departing nodes with multiple outgoing edges. We choose peer 45560 as it has the most outgoing edges (4) in the tree. After shutting down the peer (i.e., killing the process of the instance), we can see that a larger number of peers has been affected. Rebuilding the structure takes longer and some peers miss a few datagrams while others remain unaffected (e.g., 14187).

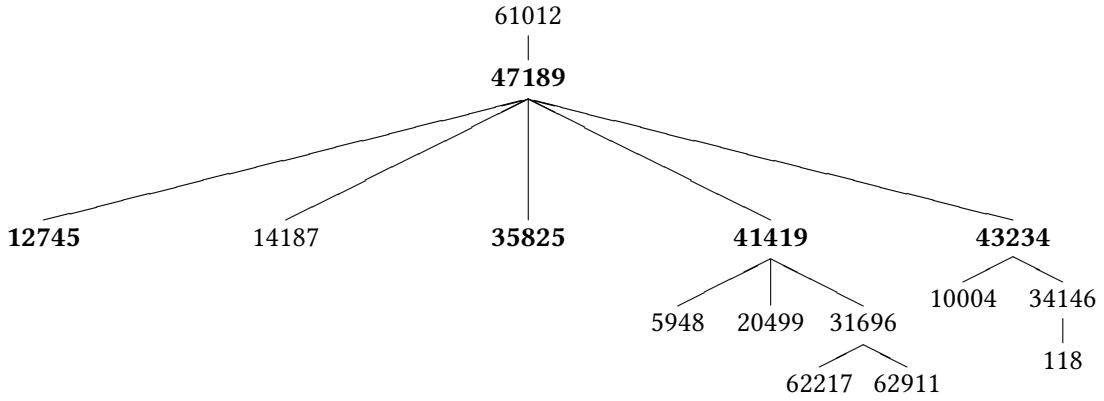


Figure 6.10: BOPscribe distribution tree after an inner node fails

Even though only neighbors of the failing nodes are directly concerned in the restructuring of the tree, a lot of nodes where affected during restructuring. We have seen that the distribution tree maintenance incorporated by BOPscribe handles both, rendezvous point and inner node

failure but a large number of peers can be affected of disruptions when peers of a high tree rank become unavailable. Dropouts result in the loss of publications and applications have to deal with the best effort delivery model.

Summary

We have shown that both cases, inner failure as well as rendezvous point failure are being mitigated very well by the tree structure. Though, we did not yet talk about the duration it takes for the tree to recover. During our first tests it took over 90 seconds to recover a failing branch after an inner node failure. This has multiple reasons: First, the Chord DHT periodically checks random peers in its finger table for failures which takes some time. Moreover, Scribe delivers content on only one path. If any of the peers in that path fails, distribution halts until the Chord rings stabilizes and, on top of that, the peers re-issue the periodic Scribe subscription. This combination explains the high dropout latencies we observed.

After adapting the maintenance duration and re-subscription values, we managed to drop the timeout to a range of $1 - 3$ seconds. It has to be noted, though, that this is only possible in LAN environment as the timeouts have to be set to higher values when operating in the public Internet. It would be interesting to compare these observations with the SplitStream and BIDIR-SAM approaches which should perform superior in such situations due to the redundant paths.

7 Flow Control and Reliability

Participants in a distributed system are often very heterogeneous with regards to network bandwidth, latency, processing power, memory capacity and many more variables. This usually leads to situations in which a sender transmits data to a receiver at a rate or of a size that the latter cannot handle (e.g., because its bandwidth is narrower or the CPU slower than the sender's). As a result, the communication between the participants is disturbed (e.g., packets are lost), which eventually degrades user experience or even leads to application failures. In a group communication scenario, the situation is even more difficult to handle, since there are multiple receivers and possibly multiple senders, that all have to agree on a mechanism to handle overwhelmed receivers. Therefore, some kind of adjustment of sending rate or packet size has to be put in place. This adjustment mechanism is called flow control. Additionally, many network applications need to assure that transmitted data actually arrived at the receiver, leveraging a reliability mechanism of the underlying transport. This chapter introduces the concepts of flow control and reliability and lays out their application to BOPlish.

Even though there is a clear distinction between flow control and congestion control, we noticed that these terms are often used as though they were interchangeable. Thus, we follow up with a differentiation of these two concepts. One source of confusion about flow and congestion control may stem from the fact that to a sender, both handle the problem of losing packets. Tanenbaum and Wetherall describe the problem scope and the difference between flow and congestion control like this:

> "An allocation problem that occurs at every level is how to keep a fast sender from swamping a slow receiver with data. Feedback from the receiver to the sender is often used. This subject is called flow control. Sometimes the problem is that the network is oversubscribed because too many computers want to send too much traffic, and the network cannot deliver it all. This overloading of the network is called congestion." [91, pp. 34-35]

By this definition, flow control focuses on end-to-end connectivity while the scope of congestion control are hop-by-hop links (see Fig. 7.1 for an illustration). One widely-known protocol

that incorporates congestion control, flow control as well as reliability is the Transmission Control Protocol (TCP). It uses a sliding window mechanism for flow control/reliability and several techniques such as slow-start and fast recovery for congestion control. With the sliding window algorithm a receiver indicates, upon reception of a TCP segment, the number of octets the sender may transmit subsequently without overwhelming the receiver [68]. Slow start – first described by Jacobson in 1988 [49] – is used to determine, how much data can be injected into the network [1].

WebRTC Data Channels rely on SCTP, which does congestion control and offers services such as ordered and reliable transmission. Sec. 7.1 gives an insight into this mechanism. It also sheds light on current flow control techniques and their application in multicast scenarios. In Sec. 7.2 we describe the flow control and reliability services of the BOPlish concept and in Sec. 7.3 we explain how we incorporated it into our reference implementation. Sec. 7.4 concludes this chapter with an evaluation.

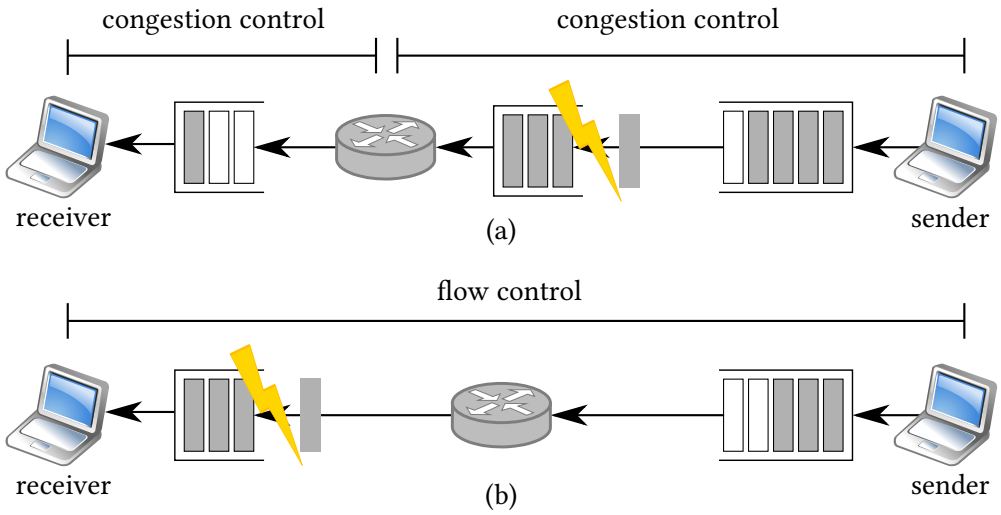


Figure 7.1: Congestion control (a) acts on the network layer to ascertain a fair use of network resources. Flow control (b) on the other hand happens between sender and receiver, e.g., when the receiver has smaller buffers than the sender.

7.1 Background and Related Work

The basic problem, that flow control ought to solve is that a sender could potentially transmit data faster than one or more receivers can process [91, pp. 201ff.]. Two commonly suggested solutions exist that approach this problem: In feedback-based (or window-based) flow control,

the receiver explicitly grants permission to the sender for transmitting packets. Protocols employing this behaviour are also called Automatic Repeat-Request (ARQ) protocols. They usually include methods for error detection (lost/corrupted packets) and recovery (in the form of data retransmission), providing a reliable transfer. A rate-based algorithm, on the other hand, suggests that sender and receiver agree on a (fixed or dynamic) packet rate.

7.1.1 Unicast Flow Control and Reliability

The simplest feedback-based flow control algorithm is stop-and-wait. Fig. 7.2 (a) illustrates the functionality of stop-and-wait. Here, the sender sends one data packet and waits for the receiver to acknowledge retrieval of that packet, indicating that the receiver is now able to process the next one. This way it is ascertained that the receiver will never be overwhelmed (as a side effect it guarantees reliable delivery). On top of that, the sender can start a timer upon transmission of every packet. Once that timer runs out, it can be assumed that the packet has been lost on the way or on the receiving side and the sender retransmits it (hence the name ARQ). The drawback of stop-and-wait is its bad performance; If the transferred packets are not very big, the throughput of such an implementation is mainly determined by the round-trip latency of the connection (in Fig. 7.2 this is the "Wait" time). The throughput will most probably never reach the link's capacity (resource under-utilization).

Thus, a sliding-window protocol is employed, that allows a sender to transmit multiple packets in a row, without having to wait for an acknowledgement from the receiver. Sliding-window protocols have three properties: they guarantee reliable delivery of packets, allow for flow control and maintain the order of packages. We focus just on the first two properties. The procedure is illustrated in Fig. 7.2 (b). Here, sender and receiver negotiate a window size, which is the number of packets (or bytes or octets, depending on the specific protocol) that the sender may transmit in a row. With high enough window sizes (depending on round-trip latency) this enables the participants to "keep the pipe full" and leverage the bandwidth at hand (a synonym for this approach is "pipelining"). To make this algorithm work without losing packets, a consecutive sequence number is typically attached to every packet. This number is included in the acknowledgement packets by the receiver. The term "window" derives from the fact that the sender maintains a window which contains all the packets that have been sent but not yet acknowledged. Whenever an acknowledgement arrives, the next packet is sent out and the window is advanced by one packet on the input stream. Apart from better resource utilization, this approach lets packets inside of the window arrive out-of-order, because they are buffered until the window can be advanced.

In error cases, however, this protocol may lead to inefficiencies, too. Suppose, a window

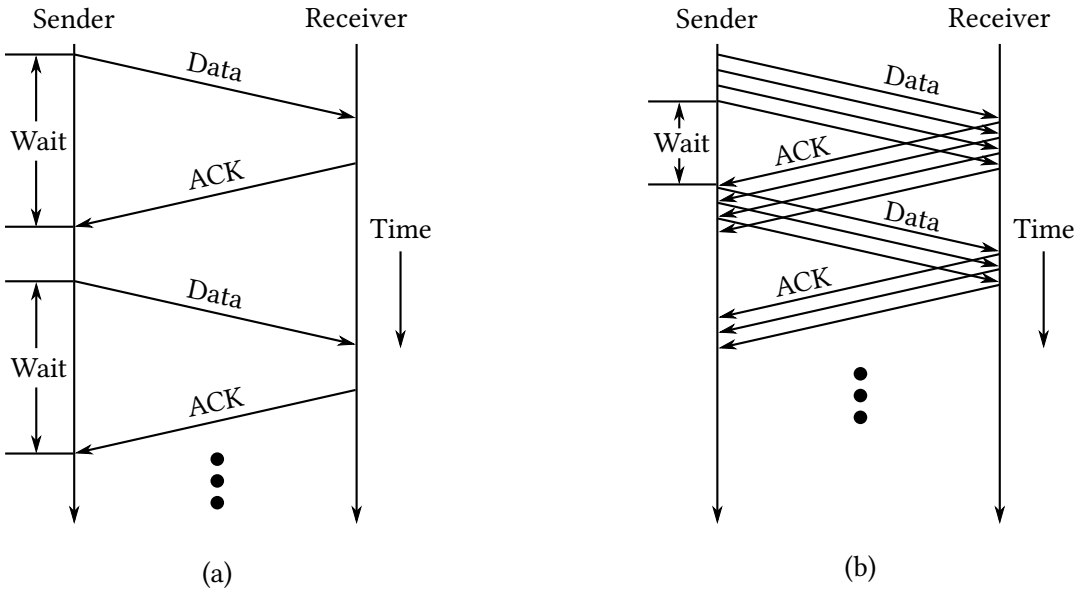


Figure 7.2: Illustration of two different feedback-based flow control algorithms. Stop-and-wait (a) is the simplest one, where every packet must be acknowledged by the receiver. With a sliding-window approach (b), the sender may transmit a certain amount of data before it has to wait for an acknowledgement (graphic based on [9, p. 503ff.]).

size of 4 is used and all 4 packets have been sent out, waiting to be acknowledged. Before the retransmission timer expires, acknowledgements for packets 2, 3 and 4 arrive, but not for packet 1. This means, that the sending window cannot be advanced until either an acknowledgement for packet 1 arrives or its timer expires. Thus, the sender is blocked at that point. On top of that, it is unclear what the sender should do after the timer expires. The options are that it transmits all packets in the current window again (perhaps sending duplicates to the receiver) or just packet 1 (which is inefficient).

One very primitive way to handle this situation is called "go-back-n". The receiver sends cumulative acknowledgements, i.e., an acknowledgement for packet 4 implies that packets 1, 2 and 3 are acknowledged, too. So, in the above situation, the sender would receive an acknowledgement for packet 4 and could advance the sending window by 4, transmitting the next 4 packets. If the acknowledgement didn't arrive in time, the sender sends all packets in its sending window again, "going back 4". Selective acknowledgements improve this cumulative acknowledgement mechanism by informing the sender about the specific packet that have been received, resulting in fewer retransmissions.

What can be seen from the described scenarios is that the parameters for any one of the used

protocols must be chosen wisely. A too large window, e.g., would lead to many packets being retransmitted in lossy networks. A too small window would result in resource under-utilization. The timeouts on the sender side also have a huge impact on performance: Too small timeouts result in many (possibly unnecessary) retransmissions; if the timeouts are set too high, both sender and receiver would possibly stay idle many times. An important metric that can be employed for finding optimal window size and timeout values is the bandwidth-delay product. It is calculated as the product of the link's capacity (in bits/second) and the round-trip delay (in seconds) and represents the maximum number of bits that can reside on the link without being acknowledged by the receiver. If this value is very high, the sliding window must also be large. [11] gives a deeper insight into the problem space and how it is solved for TCP, especially in so-called long, fat networks with a high bandwidth-delay product. An advantage of the named feedback-based approaches over rate-based ones is that – additional to offering flow control – they come with a way to handle transmission errors.

The second popular solution, rate-based flow control, is much simpler: Sender and receiver simply negotiate a rate (e.g., packets per second) at which the sender may transmit packets. From then on, no further feedback is necessary (but can very well be included in the protocol for reliability assurances or rate adaption). In [91], Tanenbaum and Wetherall promote rate-based flow control because in high-speed networks "due to the (relatively) long delay loop, feedback should be avoided: it takes too long for the receiver to signal the sender." More sophisticated approaches to rate-based flow control introduce upper bounds to the sending rate [9, p. 510].

The decision whether to use feedback-based or rate-based flow control depends highly on the use cases and the underlying network infrastructure. Tab. 7.1 summarizes the varying properties that different use cases demand of the transport: If the transport is mainly used to transmit real-time events (small packets), it is important to have low latency but the bandwidth does not play an important role. With real-time audio/video content (e.g., conferences), on the other hand, both, low latency as well as high bandwidth are necessary. The third case is the distribution of rather large data that is not time-critical (e.g., file sharing); it demands a high bandwidth and does not react too bad on high latencies between sender and receiver.

Usage	Low Latency	High Bandwidth	Reliable
Real-time: Audio/Video	yes	yes	no
Content distribution	no	yes	yes
Real-time: Events	yes	no	yes

Table 7.1: Different usages require different flow control and reliability approaches.

7.1.2 WebRTC Congestion/Flow Control and QoS

The Data Channel specification [53] offers only little insight into the specific congestion and flow control mechanisms employed by WebRTC implementations. The following enumeration lists mentions of congestion control in the specification:

- "Data channels of a PeerConnection MUST be congestion controlled; either individually, as a class, or in conjunction with the SRTP media streams of the PeerConnection, to ensure that data channels don't cause congestion problems for these SRTP media streams, and that the WebRTC PeerConnection does not cause excessive problems when run in parallel with TCP connections."
- "The important features of SCTP in the WebRTC context are: [...] Usage of a TCP-friendly congestion control. [...] The congestion control is modifiable for integration with the SRTP media stream congestion control."
- "SCTP provides congestion control on a per-association base. This means that all SCTP streams within a single SCTP association share the same congestion window. Traffic not being sent over SCTP is not covered by the SCTP congestion control. Using a congestion control different from than the standard one might improve the impact on the parallel SRTP media streams."

Flow control is not mentioned at all and with regard to congestion control, the specification refers to other Internet standards. To gain an insight into the congestion control mechanisms, it is thus necessary to inspect the SCTP specification. SCTP as defined in [87] mandates both flow and congestion control similar to the mechanisms employed by TCP. Specifically, these mechanisms are standardized in [3] (slow start, congestion avoidance, fast retransmit, fast recovery) and [12] (avoidance of "silly window syndrome"). Both major Data Channel-enabled WebRTC implementations – Chrome and Firefox – ship with this default mechanism. Currently, handling congestion control of Data Channels together with media streams is discussed at the IETF in the Rtcweb working group as well as the RMCAT (RTP Media Congestion Avoidance Techniques) working group[1]. To date, there are no proposed drafts published by either of these groups, which suggests that implementations will not change the current behaviour dramatically in the foreseeable future.

Theoretically, messages passed to `RTCDataChannel.send()` can be arbitrarily large and the implementation handles fragmentation and re-assembly. Currently, though, only Firefox

[1] `http://tools.ietf.org/wg/rmcat/`

implements fragmentation properly. In Chrome, passing packets larger than the employed SCTP library's default of 64 KBytes/packet will result in an error condition. This is most likely to change in the future (mostly depending on the finalization of the NDATA draft [88]), but currently, applications will have to account for this limitation and implement fragmentation and re-assembly by themselves.

Send/Receive Buffering

The WebRTC specification [7] defines a read-only attribute of the `RTCDataChannel` interface, named `bufferedAmount`. This attribute is one potential parameter that can be employed for implementing overlay flow control. It indicates, how much of the data that has been passed to `RTCDataChannel.send()` is currently being buffered at the sender. If the value of this attribute increases constantly, it is an indication of congestion and the sending rate should probably be decreased. We investigated the usability of this attribute for our purposes of flow control and examined both the Chrome browser as well as Firefox. Both employ a custom buffer in their Data Channel implementations. This buffer is completely independent of the underlying SCTP's buffer. Firefox has no built-in upper limit of buffered data[2,3], while Chrome has a hard-coded limit of 16 MBytes[4] defined in the variable `kMaxQueuedSendDataBytes`. One pitfall of Chrome's buffering implementation is that, if the amount of buffered data in the Data Channel exceeds `kMaxQueuedSendDataBytes`, the channel is immediately closed. This behaviour mandates a careful monitoring of `bufferedAmount` and eventually stopping of calling `RTCDataChannel.send()`, so that the channel does not get closed.

As for a receive buffer, the situation is similar. Firefox has no hard-coded buffer limit while Chrome has a variable `kMaxQueuedReceivedDataBytes` that is set to 16 MBytes. If this 16 MByte buffer is exceeded (e.g., by the sender overwhelming the receiver with data), the connection is closed by the client's browser. The application has no means of anticipating such a situation because the WebRTC API does not define an attribute that would represent the receive buffer. This problem is relaxed, though, by the fact, that `kMaxQueuedReceivedDataBytes` is only filled as long as no `onmessage` handler has been set on the Data Channel. For Firefox, the situation is similar: The Firefox WebRTC implementation fires a `message` event as soon as data arrives on the SCTP connection. Thus, the limiting factor would be the browser's event

[2] `https://mxr.mozilla.org/mozilla-central/source/netwerk/sctp/datachannel/DataChannel.cpp#2227`
[3] `https://mxr.mozilla.org/mozilla-central/source/netwerk/sctp/datachannel/DataChannel.cpp#1041`
[4] `https://code.google.com/p/webrtc/source/browse/trunk/talk/app/webrtc/datachannel.cc#39`

queue size. In contexts different from browser applications (e.g., system application written in C or C++), developers would typically implement a dispatching functionality with a dedicated thread for buffering incoming messages and dispatching them to a processing thread. This prevents the network buffer to fill up. In Web applications, a Web Worker could be used for dispatching [45]. Unfortunately, no browser implementation currently allows for the usage of Data Channels from a worker context.

7.1.3 Multicast Flow Control

In simple WebRTC-based scenarios, where data is transported from one peer to another using a Data Channel, the built-in hop-by-hop flow and congestion control mechanisms as outlined above suffice to prevent the receiver from being overwhelmed with data. When a transport is run over a network of WebRTC peers, though, an application-level transport protocol must be employed to assure that receivers are not overwhelmed with huge data bursts. Flow control algorithms face the major challenge of scalability when applied to multicast networks [63]. Algorithms must be scalable in the sense that feedback from receivers arrives at the sender in a timely fashion (i.e., before messages get lost). Aggregated feedback is important because in multicast scenarios, all senders may provide flow control feedback (e.g., rate adaption requests) to the senders. Such feedback messages can easily overwhelm intermediate peers as well as the multicast senders (an effect known as feedback implosion) and hence must be synchronized/aggregated at some point on their way to the senders.

Since the underlay transport of Data Channels already offers hop-by-hop congestion and flow control, it seems obvious to incorporate these into the overlay flow control. Several publications have analyzed the impact of leveraging the congestion and flow control of underlying transports (mostly TCP) in overlay networks. Most of the proposed approaches found in the literature focus on single-source multicast. One specific problem of leveraging the hop-by-hop congestion control is that slow receivers will slow down the sender [18]. This is the case in the simplest (and not scalable) technique: Every receiver sends flow control information towards the sender (end-to-end multicast flow control). This can be achieved simply in the form of acknowledgements for every processed packet. The sender only transmits the next packet, if it has received acknowledgements for the previously sent one. This resembles the stop-and-wait ARQ protocol.

In [95], Urvoy-Keller and Biersack propose a backpressure algorithm (Overlay Multicast Congestion Control (MCC)), so that flow control is conducted between adjacent nodes (leveraging underlay congestion control and solving the problem of feedback synchronization) instead of between all receivers and the sender (end-to-end). The idea is that each node in a multicast tree has one input buffer and N output buffers (where N is the number of downstream links

on the tree). Incoming packets are only copied to the output buffers if all output buffers have enough capacity. This way, flow control between input and output buffers is achieved. Still, congestion on one or more of the output buffers has to propagate back to the upstream peer. The authors state that "to ensure a backpressure up to the source, the flow and congestion control mechanism must be coupled." For this to work, the input buffer is limited and received packets are only ACKed, if they have been copied to all output buffers. Urvoy-Keller and Biersack prove that this model has better performance than an end-to-end approach, especially with regards to an increasing number of recipients.

7.2 A Concept for Flow Control and Reliability in BOPlish

We now describe our approach to add flow control as well as reliability services to BOPlish. The requirements that we identified as crucial are as follows: BOPlish applications must not be overwhelmed by data sent from a remote peer (a central aspect of flow control). Depending on the use case, applications may allow for large receive buffers so that bursts of incoming data are buffered and can be processed later on. On the other hand, real-time applications may not want to buffer huge amounts of data, but rather signal the sender to stop sending or lower its transmit rate. The second requirement is that applications must be able to choose between reliable and unreliable data transfer, also depending on the use case. Since BOPlish is designed to offer a generic transport layer with different quality of service assurances, an API must exist so that applications built on top of it can directly communicate their requirements to BOPlish.

Adding a generic reliable transport to an multi-hop overlay network is considerably complex. It is much simpler to incorporate mechanisms that are specifically tied to certain reliability requirements. We identified two different use cases building on the requirements mentioned above that we integrated into BOPlish: Delay-sensitive, real-time applications (such as an A/V conference) and fully reliable applications (such as file sharing). Those resemble rows 1 and 2 of Tab. 7.1. While the first type of applications accepts possible message drops, the latter one must accomplish for all messages to be delivered, even at the cost of transfer delay. Both naturally require receivers to be able to control the flow rate. We extended the BOPlish API with a set of methods and parameters so that developers can choose the reliability assurances of their group communication application.

This extended API is shown in Lst. 7.1. The API calls `registerProtocol` and `route` (see Sec. 4.4.1 and Sec. 4.4.4, respectively) were extended by a JSON object called `reliabilityOptions`. Using this object, developers instruct BOPlish to apply the given restrictions on each transferred packet of that specific protocol. The same applies to the `buffers` object,

where developers instruct BOPlish to only accept M packets from the upper-layer application, without receiving an acknowledgement from the remote side. Only N packets are accepted in-bound, that are not processed by the upper-layer application. Fig. 7.3 shows how this looks like for a single peer. Here, application 1 has a send buffer of size 4 and a receive buffer of size 4. Application N has a send buffer of 2 and a receive buffer of size 5. Further details on the semantics of the options are explained in 7.3.

```
1 BOPlishClient.registerProtocol(identifier, reliabilityOptions,
      buffers)
2 Router.route(to, message, callback, reliabilityOptions)
3
4 reliabilityOptions = {
5    reliable: [true|false], // default: true
6    timeout: TIMEOUT_IN_MS // default: calculated from RTT
7    maxRetransmits: N // default: 0
8 }
9
10 buffers = {
11    sendBufferSize: M // default: unlimited
12    recvBufferSize: N // default: unlimited
13 }
```

Listing 7.1: The new API calls, extended by options for reliability assurances as well as buffer sizes.

7.3 Implementation

Reliability and flow control in BOPlish are services that application developers request by using the API of the BOPlishClient interface defined in Sec. 7.2. For this to work, we had to slightly adapt the transport message format (Sec. 4.3.3) as well as the inner workings of our Chord implementation.

7.3.1 Unicast Reliable Transfer

The Chord implementation described in Sec. 4.5 offers an API method `route(to, message, callback)`, that routes a message to the peer responsible for the ID `to`. The remote peer acknowledges reception of the message with an ACK message and `callback` is called on the sending peer (`route()` is always non-blocking). In the process of adding a reliability service

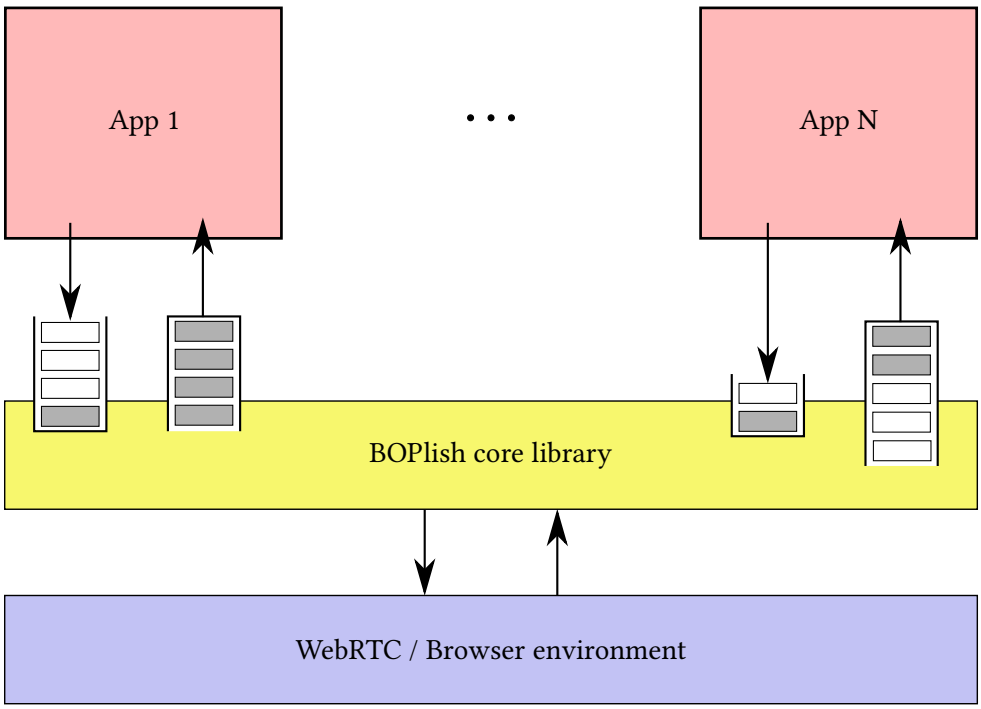


Figure 7.3: The general flow control architecture of a single BOPlish instance. Every application has a dedicated send and receive buffer.

to the DHT routing, we added the `reliabilityOptions` parameter from Lst. 7.1 to this method.

Upon calling `route()`, developers instruct Chord on the reliability requirements they would like to be applied. If `reliable` is set to `true`, the receiving peer sends back an ACK message. If this message arrives at the sender, the callback is called and the transmission is deemed successful. If the message does not get acknowledged by the receiver after `timeout` milliseconds, the transmission is deemed unsuccessful and `callback` is invoked with an error message so that the application can react appropriately. The option `maxRetransmits` controls the behaviour when the timer expires: If `maxRetransmits` is an integer value N greater than 0, then Chord tries to deliver the message N times, each time restarting the timer specified in `timeout`. Thus, the delivery is deemed unsuccessful after $timeout * maxRetransmits$ milliseconds of not receiving an acknowledgement. We exposed this transmission control feature to applications by adding the same option parameter to the `registerProtocol(identifier)` method defined in Sec. 4.4.1. This way, developers of BOPlish applications can make use of the reliability extension on a per-protocol basis.

The `timeout` value is especially hard to set correctly because of the varying latencies between different peers in the DHT. Given a round-trip time (RTT) of N milliseconds, it is not advisable to set the value of this parameter lower than N (the time for the data packet to arrive at the receiver plus the time for the ACK packet to arrive at the sender). The processing time at the receiver must also be added to get a meaningful value. Thus, our Chord implementation maintains RTT values for every peer it is communicating with and takes this value into account as default, when the application developers has not provided this option.

To implement this behaviour, we changed our `ChordNode` classes so that every node that a Chord instance maintains an open connection to, calculates the RTT by storing the time stamp whenever a data packet is sent to a remote peer. When an acknowledgement arrives, the time of the arrival is subtracted from the transmission time. The RTT for a remote node N thus results from the following formula:

$$RTT_N = t_{rx} - t_{tx}$$

This value is constantly updated with every transmission conducted to that node. It is easy to improve this simple RTT calculation in our code by, e.g., maintaining a smooth average, as is done with many TCP implementations. One factor that makes it difficult, though, to maintain correct RTT values is that the structure of the DHT may change often and RTTs vary extremely (as opposed to TCP, where the structure of the network does not change significantly during a session). Therefore, to simplify further research on the topic, we built the API in such a way that developers can plug-in an RTT estimator of their own into our Chord implementation. They include a `script` tag in their HTML hosting their application that points to their estimator implementation. Such implementations must implement an interface with three distinct properties: A zero-argument constructor, a function `newRTT(rtt)`, taking the value of the currently measured RTT as parameter as well as a function `rto()` that returns the RTO for the next transmission. Plugging this code into our Chord implementation works like this:

```
1 ChordNode.RTTestimator = MyEstimator;
```

A reference implementation of such an estimator – listed in Lst. 7.2 – is included in BOPlish and used as default. It is shown in and calculates the next RTO by taking into account the last 10 measured RTTs, building a simple average and adding a factor of 2 (as is done for TCP, too).

```
1 ChordNode.RTTestimator = function() {
2   this._RTO = peerConfig.messageTimeout || 1000;
3   this._history = new RingBuffer(10);
```

```
4     this.maxRTT = -1;
5 };
6
7 ChordNode.RTTestimator.prototype = {
8     /**
9      * Feeds a new calculated RTT value into the estimator for
10     * further processing.
11     */
12    newRTT: function(rtt) {
13       this._history.push(rtt);
14       if (rtt > this.maxRTT) {
15          this.maxRTT = rtt;
16       }
17       var histArr = this._history.getall();
18       this._RTO = (histArr.reduce(function(prev, cur) {
19          return prev + cur;
20       }, 0) + this.maxRTT) / (histArr.length + 1);
21    },
22
23    /**
24     * Retrieves the current calculated RTO.
25     */
26    rto: function() {
27       return 2*this._RTO;
28    }
29 };
```

Listing 7.2: RTT estimator using the average of the last 10 RTT values to calculate an RTO.

7.3.2 Unicast Flow Control

Additional to reliable transmission, we added the possibility to specify a buffer size per BOPlish protocol instance. This makes it possible for developers to assure, only a certain number of unprocessed messages remains in the network. To support per-protocol buffers, we added the parameter `buffers` to method `registerProtocol()`, which denotes the size of the send and receive buffer in number of messages. When the application calls the `send()` method of the protocol instance returned by `registerProtocol()`, the BOPlish implementation copies the message into the send buffer. If the protocol is instructed to transmit data reliably (using the options from above), the message is extracted from the send buffer only when the

transmission is marked complete. Similar to the return values EWOULDBLOCK and EAGAIN known from the POSIX Socket API[5] , the `send()` method call of the protocol returned by `registerProtocol()` returns a value of -1 when the send buffer is full. The application can then act accordingly by, e.g., sleeping for a number of seconds and then retry.

The receive buffer works in a similar way. Messages are put into an inbound queue by the BOPlish instance and passed to the application for further processing. Only when the application function returns, the message is removed from the queue by BOPlish. If the queue is full (the number of messages in the queue has reached the configured maximum), the message is discarded and the sender receives an error message, indicating the fact. This error message contains the size of the receive buffer, so that the sending side is able to adapt its transmission rate (by, e.g., only leaving as many messages in-flight (unacknowledged) as fit in the receiver's buffer). This way, receivers are able to control the flow rate of every sender individually.

7.4 Evaluation

In order to evaluate our implemented solutions, we started by gaining insights into common transmission delays inside of BOPlish networks. We erected a BOPlish community of 17 nodes inside of a physical network with LAN as well as WLAN stations. The most simple RTT estimator, that we plugged into the system, always takes the last measured value of the RTT as RTO. Unsurprisingly, this lead to very many timeouts due to the big amount of jitter (see Sec. 5.2). What we also perceived during testing of our chat demo application was that whenever we started sending many chat messages, the round-trip times increased significantly. By conducting targeted RTT measurements in a BOPlish network of 2 peers, we were able to reproduce this finding. Fig. 7.4 shows the results of this measurement. During the non-interactive phase, only DHT maintenance messages are exchanged between the two nodes. When the user starts interacting with the application, RTTs increase to values up to 20 times higher. This observation can be explain with two effects: First, the higher message load on each node, caused by the user-generated chat messages, causes the BOPlish library to experience higher load. Second, and most significantly, the single-threaded runtime environment of JavaScript applications has a direct impact on delay times. Lst. 7.3 shows the code that handles incoming messages.

All messages are passed to the upper application layer (line 3), the application acts appropriately (e.g., by displaying the chat message in the user interface) and then returns control back to the lower Chord layer, that can then send back the ACK (for a reliable transfer). All of this,

[5] `http://pubs.opengroup.org/onlinepubs/007908799/xns/send.html`

also the processing of incoming messages, updating the user interface etc., happen inside one and the same thread of execution. Thus, if the processing of a message in the application takes a very long time, the transmission of the ACK packet is delayed, too.

```
1 _handle_response: function(msg) {
2    delete this._pending[msg.seqnr];
3    this._pending.callback(msg.error, msg);
4 },
```

Listing 7.3: The Chord code handling incoming messages

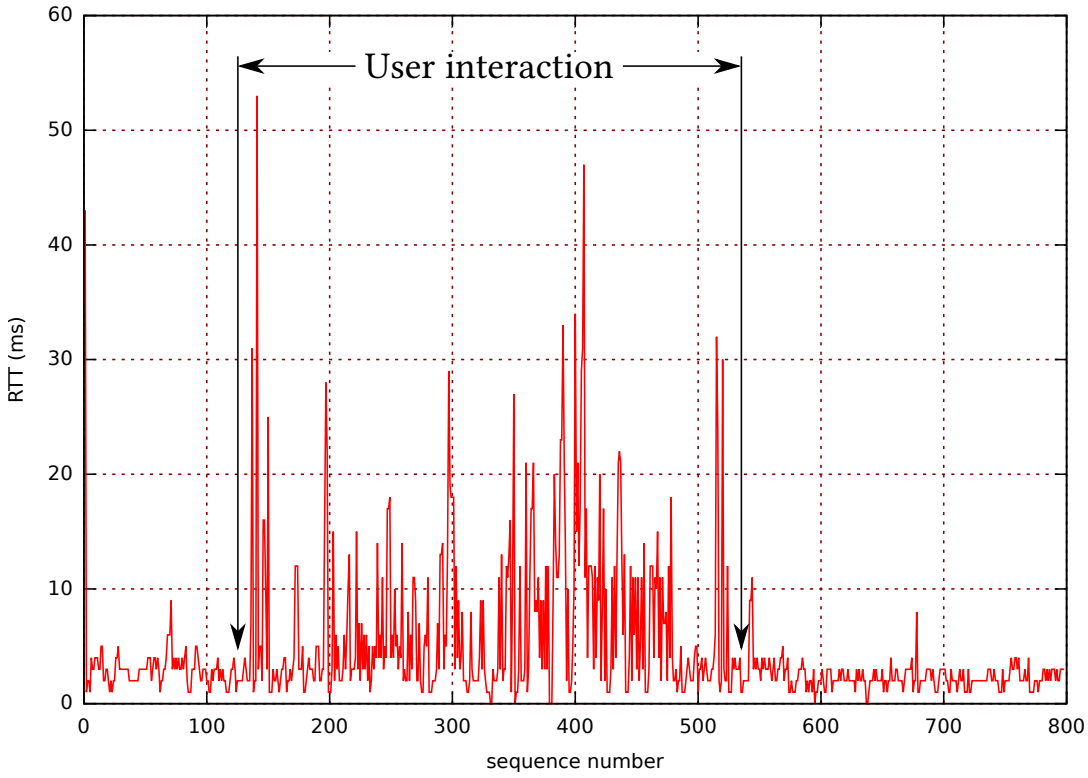


Figure 7.4: RTT values (in ms) in a network of 2 BOPlish peers. At about sequence number 120, we started a burst of user interaction causing many messages to be transmitted. At about 520, we stopped the interaction. It can be seen that during the interaction, the delay between the 2 peers increased significantly.

As a second step, we investigated further on the RTTs in the 17-node community to get insights into what we can expect as delays in a common scenario. Fig. 7.5 shows the cumulative distribution function of round-trip delays after sending about 150,000 messages through the community. As can be seen, around 90% of the delays are under 107ms. Since there are huge

outliers, though, a good RTT estimator implementation for BOPlish must take into account big variances.

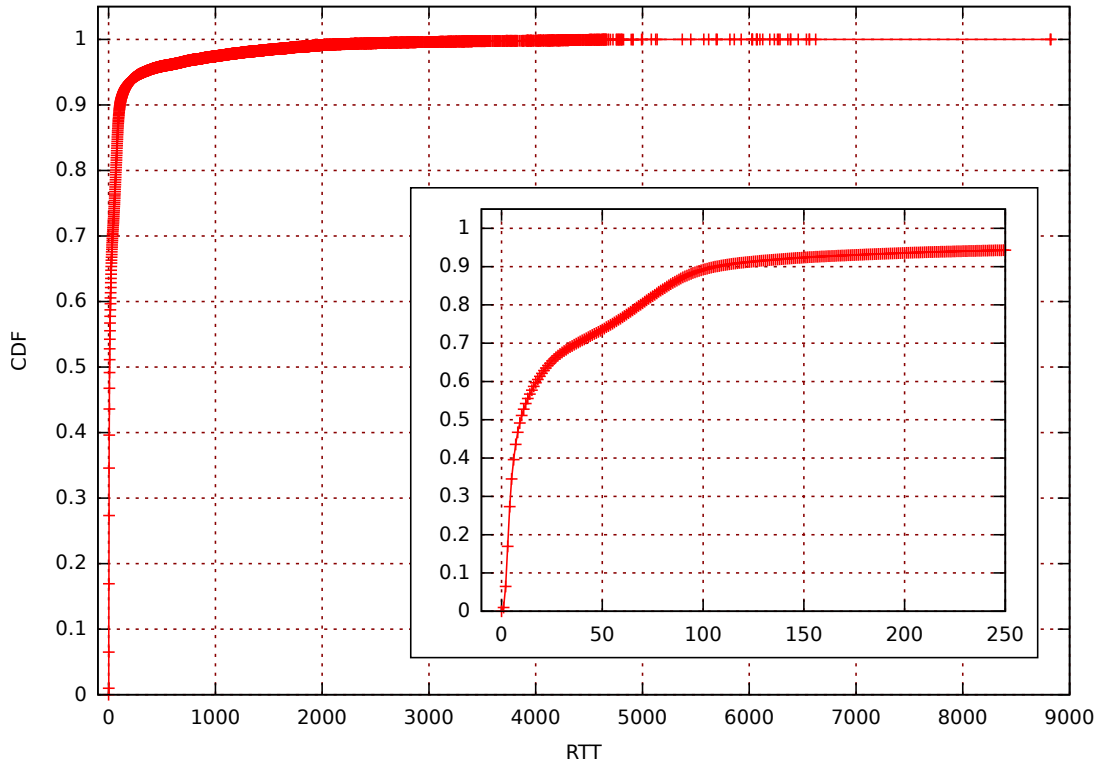


Figure 7.5: Cumulative distribution function of round-trip times of 151,622 reliable transmissions in a BOPlish network with 17 nodes. 90% of RTTs are under 107ms. It can be seen, though, that there are huge outliers with values greater than 8800ms.

7.5 Outlook

We implemented a first prototype of flow control and reliability extensions for BOPlish. These are designed explicitly to enable extension of the mechanisms by third party developers. Currently, applications dictate send/receive buffer sizes to BOPlish. In a next iteration of the concept, it makes sense to investigate possibilities of BOPlish centrally managing buffers and access to them, e.g. by a round-robin mechanism. This way, BOPlish could guarantee every application running on one peer to have fair access to the community resources.

Our evaluation showed that round-trip times vary extremely over time of usage of the network. Thus, a deeper exploration of retransmission timeout possibilities (RTO) is well

worth it. The adaptation of the TCP mechanisms (e.g, smooth average over past RTTs) or more sophisticated approaches such as the ones recommended by Rhea et al. in [74] can easily be experimented with, by using the plug-in mechanism for RTT estimators. It is also worthwhile investigating a separation of application-level message processing from the underlying BOPlish transport layer. We will investigate the possibility to separate all application-level code in a Web worker so that the application handles incoming messages without blocking the transport layer code.

With regards to multicast, we only outlined the concept that we envision for BOPlish. In this work, we rather focused on our multi-hop unicast flow control and reliability solution. Nevertheless, multicast flow control is an important building block of solid group communication frameworks. We suppose that adding single-source multicast reliability and flow control to BOPlish is an interesting next step. The limitation of focussing on SSM should not impose great performance or functionality issues on applications, since in our Scribe-based multicast communication, all messages are sent via one root node per group, thus resembling an SSM. The approach to multicast reliability, that we find feasible of integrating easily into BOPlish borrows from the push-back flow control solution (also known as backpressure) suggested by Urvoy-Keller and Biersack [95]. The authors assume the overlay network to be run atop a reliable network, where each node is connected to other nodes via TCP. This resembles the BOPlish network, where nodes are also connected to each other via reliable transports (SCTP). For applications with predictable or minimum flow rate requirements, a rate-based flow control mechanism would possibly be feasible, too.

8 Conclusions/Future Work

An efficient, name-based way of publishing content on top of the near-everywhere available Web platform is the main contribution of this thesis. During our work on the subject, we continuously published our ideas and discussed them with the research community [98, 97, 104]. Our work showed the conceptualization, implementation and evaluation of BOPlish, a user-centric approach to distributed content communities in Web browsers. We described our ideas by applying a bottom-up approach, from browser-based DHTs over user-centric content identifiers to multicast group communication and quality of service assurances. We validated our idea by implementing all designed aspects in a JavaScript library and building real-world applications on top of that library. Additionally, measurements were conducted regarding performance and stability properties of our implementation. By releasing all the output of our work on GitHub (https://github.com/boplish), we let interested researchers and developers investigate the code and the underlying concepts.

The consolidation of Web-based technology and P2P paradigms will prospectively result in similar approaches like BOPlish. During our work, a wide amount of thematic ground has been touched. The result is a vertical cut-through that shows the general feasibility of our approach and a wide range of possibilities for future extensions. Questions remain regarding technical aspects, e.g., other DHT mechanisms to stabilize the overlay network or enhance performance on the group communication layer. We imagine large-scale, distributed applications running on the Web without the need of centralized infrastructure. It remains to be investigated how such applications (e.g., Distributed Social Networks) behave on BOPlish. One of the next steps in clearing the way is to validate the conceptual assumptions in a widely distributed emulation that reflects the Web more closely than a local environment. It should be noted that such tests should continuously be conducted while the WebRTC implementations and tooling are advanced. Additionally, Web platform standards, such as the Web Cryptography API, pave the way to, e.g., enhance the privacy and security of BOPlish applications. Another aspect that arose during the work on BOPlish is to couple a Web-based group communication mechanism with other multicast islands. Such a hybrid multicast may be implemented in a stand-alone library and bring any form of multicast to the massively available Web-capable devices.

Bibliography

[1] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, IETF, September 2009.

[2] Mark Allman. Personal Namespaces. In *Proc. of the 6th ACM Workshop on Hot Topics in Networks (HotNets-VI)*, New York, NY, USA, 2007. ACM.

[3] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control. RFC 2581, IETF, April 1999.

[4] Harald Alvestrand. Overview: Real Time Protocols for Browser-based Applications. Internet-Draft – work in progress 13, IETF, November 2014.

[5] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711, IETF, March 2004.

[6] 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)*, pages 19–24, 2012.

[7] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. WebRTC 1.0: Real-time Communication Between Browsers. W3C Working Draft, World Wide Web Consortium, 2013.

[8] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, IETF, January 2005.

[9] D.P. Bertsekas and R.G. Gallager. *Data networks*. Prentice-Hall, 1987.

[10] Stefan Birrer and Fabian E. Bustamante. The Feasibility of DHT-based Streaming Multicast. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 288–298, Washington, DC, USA, 2005. IEEE Computer Society.

[11] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. TCP Extensions for High Performance. RFC 7323, IETF, September 2014.

[12] Robert Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, IETF, October 1989.

[13] Tom Callahan, Mark Allman, Michael Rabinovich, and Owen Bell. On Grappling with Meta-information in the Internet. *SIGCOMM Comput. Commun. Rev.*, 41(5):13–23, October 2011.

[14] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure Routing for Structured Peer-to-peer Overlay Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314, December 2002.

[15] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony I. T. Rowstron, and Atul Singh. SplitStream: High-Bandwidth Content Distribution in Cooperative Environments. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II. Second International Workshop, IPTPS 2003 Berkeley, CA, USA, February 21-22, 2003 Revised Papers*, volume 2735 of *LNCS*, pages 292–303, Berlin Heidelberg, 2003. Springer–Verlag.

[16] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, 2002.

[17] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer Overlays. In *Proceedings of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2003)*, volume 2, pages 1510–1520, Washington, DC, USA, 2003. IEEE Computer Society.

[18] A. Chaintreau, F. Baccelli, and C. Diot. Impact of Network Delay Variations on Multicast Sessions with TCP-like Congestion Control. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 1133–1142 vol.2, 2001.

[19] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[20] M. Cotton, L. Vegoda, and D. Meyer. IANA Guidelines for IPv4 Multicast Address Assignments. RFC 5771, IETF, March 2010.

[21] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A Decentralized Network Coordinate System. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, August 2004.

[22] Frank Dabek, Ben Y. Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003, Revised Papers*, volume 2735 of *LNCS*, pages 33–44, Berlin Heidelberg, 2003. Springer–Verlag.

[23] David Dahl and Ryan Sleevi. Web Cryptography API. W3C Working Draft, World Wide Web Consortium, 2013.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of the 21th ACM Symposium on Operating Systems Principles (SIGOPS'07)*, pages 205–220, New York, NY, USA, 2007. ACM.

[25] S. E. Deering and D. R. Cheriton. Host groups: A multicast extension to the Internet Protocol. RFC 966, IETF, December 1985.

[26] Steve Deering. Host extensions for IP multicasting. RFC 1112, IETF, August 1989.

[27] John R. Douceur. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[28] Peter Druschel et al. FreePastry. http://freepastry.rice.edu/FreePastry/, 2008.

[29] Deborah Estrin, Dino Farinacci, Ahmed Helmy, David Thaler, Stephen Deering, Mark Handley, Van Jacobson, Ching gung Liu, Puneet Sharma, and Liming Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. RFC 2117, IETF, June 1997.

[30] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[31] S. Farrell and H. Tschofenig. Pervasive Monitoring Is an Attack. RFC 7258, IETF, May 2014.

[32] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

[33] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. Evading Censorship with Browser-based Proxies. In *Proceedings of the 12th international conference on Privacy Enhancing Technologies*, PETS'12, pages 239–258, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] Ulrich Gallersdörfer. Infrastructures of browser-based botnets. Bachelor's thesis, Technische Universität München, October 2014.

[35] Ali Ghodsi, Teemu Koponen, Jarno Rajahalme, Pasi Sarolahti, and Scott Shenker. Naming in Content-oriented Architectures. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, ICN '11, pages 1–6, New York, NY, USA, 2011. ACM.

[36] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly & Associates, 2013.

[37] Mark Gritter and David R. Cheriton. An Architecture for Content Routing Support in the Internet. In *Proc. USITS'01*, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.

[38] Philipp Hagemeister. Censorship-resistant Collaboration with a Hybrid DTN/P2P Network. Master's thesis, HHU Düsseldorf, March 2012.

[39] Yang Han, K. Koyanagi, T. Tsuchiya, T. Miyosawa, and H. Hirose. A Trust-based Routing Strategy in Structured P2P Overlay Networks. In *2013 International Conference on Information Networking (ICOIN)*, pages 77–82, Jan 2013.

[40] M. Handley. Why the Internet Only Just Works. *BT Technology Journal*, 24(3):119–129, Jul. 2006.

[41] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566, IETF, July 2006.

[42] M. Handley, I. Kouvelas, T. Speakman, and L. Vicisano. Bidirectional Protocol Independent Multicast (BIDIR-PIM). RFC 5015, IETF, October 2007.

[43] Ian Hickson. Server-Sent Events. W3C Candidate Recommendation, World Wide Web Consortium, 2012.

[44] Ian Hickson. The WebSocket API. W3C Candidate Recommendation, World Wide Web Consortium, 2012.

[45] Ian Hickson. Web Workers. W3C Candidate Recommendation, World Wide Web Consortium, May 2012.

[46] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. HTML5. W3C Proposed Recommendation, World Wide Web Consortium, September 2014.

[47] Robert M. Hinden and Stephen E. Deering. IPv6 Multicast Address Assignments. RFC 2375, IETF, July 1998.

[48] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation, World Wide Web Consortium, April 2004.

[49] V. Jacobson. Congestion Avoidance and Control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988.

[50] Van Jacobson, Diana K. Smetters, James D. Thornton, and Michael F. Plass. Networking Named Content. In *Proc. of the 5th Int. Conf. on emerging Networking EXperiments and Technologies (ACM CoNEXT'09)*, pages 1–12, New York, NY, USA, December 2009. ACM.

[51] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne. REsource LOcation And Discovery (RELOAD) Base Protocol. RFC 6940, IETF, January 2014.

[52] Cullen Jennings, Bruce Lowekamp, Eric Rescorla, Salman Baset, Henning Schulzrinne, and Thomas Schmidt. A SIP Usage for RELOAD. Internet-Draft – work in progress 13, IETF, July 2014.

[53] Randell Jesup, Salvatore Loreto, and Michael Tuexen. WebRTC Data Channels. Internet-Draft – work in progress 12, IETF, September 2014.

[54] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A Data-Oriented (and beyond) Network Architecture. *SIGCOMM Computer Communications Review*, 37(4):181–192, 2007.

[55] Gunnar Kreitz and Fredrik Niemelä. Spotify – Large Scale, Low Latency, P2P Music-on-Demand Streaming. In *Proc. of the 10th IEEE International Conference on Peer-to-Peer Computing (P2P'10)*, Aug 2010.

[56] Dirk Kutscher, Suyong Eum, Kostas Pentikousis, Ioannis Psaras, Daniel Corujo, Damien Saucez, Thomas C. Schmidt, and Matthias Wählisch. ICN Research Challenges. IRTF Internet Draft – work in progress 02, IRTF, February 2014.

[57] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

[58] R. Marques and A. Zuquete. User-Centric, Private Networks of Services. In *2013 International Conference on Smart Communications in Network Technologies (SaCoNeT)*, volume 01, pages 1–5, June 2013.

[59] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of the 1st Int. Workshop on Peer-to Peer Systems (IPTPS '02)*, pages 53–65, Cambridge, MA, USA, 2002.

[60] A. J. Meyn. Browser to Browser Media Streaming with HTML5. Master's thesis, Technical University of Denmark, DTU Informatics, Kgs. Lyngby, Denmark, 2012.

[61] John Moy. Multicast Extensions to OSPF. RFC 1584, IETF, March 1994.

[62] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Operating System Review*, 44(3):2–19, August 2010.

[63] K. Obraczka. Multicast Transport Protocols: A Survey and Taxonomy. *Communications Magazine, IEEE*, 36(1):94–102, Jan 1998.

[64] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An Application Level Multicast Infrastructure. In *Proc. of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems (USITS'01)*, pages 49–60, Berkeley, CA, USA, 2001. USENIX Association.

[65] Kostas Pentikousis, Borje Ohlman, Daniel Corujo, Gennaro Boggia, Gareth Tyson, Elwyn Davies, Antonella Molinaro, and Suyong Eum. Information-centric Networking: Baseline Scenarios. Internet-Draft – work in progress 03, IETF, August 2014.

[66] S. Perreault and J. Rosenberg. Traversal Using Relays around NAT (TURN) Extensions for TCP Allocations. RFC 6062, IETF, November 2010.

[67] Peter Pietzuch, David Eyes, Samuel Kounev, and Brian Shand. Towards a Common API for Publish/Subscribe. In *Proc. of the Inaugural International Conference on Distributed Event-based Systems (DEBS'07)*, pages 152–157, 2007.

[68] Jon Postel. Transmission Control Protocol. RFC 793, IETF, September 1981.

[69] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-Level Multicast Using Content-Addressable Networks. In Jon Crowcroft and Markus Hofmann, editors, *Proc. of 3rd Intern. Workshop on Network Group Communication (NGC'01)*, volume 2233 of *LNCS*, pages 14–29, London, UK, Nov. 2001. Springer–Verlag.

[70] Sylvia Paul Ratnasamy. *A Scalable Content-Addressable Network*. PhD thesis, University of California, Berkeley, October 2002.

[71] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, IETF, January 2012.

[72] Eric Rescorla. Security Considerations for WebRTC. Internet-Draft – work in progress 07, IETF, July 2014.

[73] Eric Rescorla. WebRTC Security Architecture. Internet-Draft – work in progress 10, IETF, July 2014.

[74] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[75] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, IETF, April 2010.

[76] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, IETF, October 2008.

[77] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264, IETF, June 2002.

[78] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, IETF, June 2002.

[79] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 2218 of *LNCS*, pages 329–350, Berlin Heidelberg, November 2001. Springer–Verlag.

[80] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The Design of a Large-Scale and Event Notification Infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication. Third International COST264 Workshop, NGC 2001. Proceedings*, volume 2233 of *LNCS*, pages 30–43, Berlin Heidelberg, 2001. Springer–Verlag.

[81] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov 1984.

[82] Vinay Setty, Gunnar Kreitz, Roman Vitenberg, Maarten van Steen, Guido Urdaneta, and Staffan Gimåker. The Hidden Pub/Sub of Spotify: (Industry Article). In *Proc. of the 7th ACM International Conference on Distributed Event-based Systems (DEBS'13)*, pages 231–240, New York, NY, USA, 2013. ACM.

[83] R. Shirey. Internet Security Glossary, Version 2. RFC 4949, IETF, August 2007.

[84] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 261–269, London, UK, UK, 2002. Springer-Verlag.

[85] Diana Smetters and Van Jacobson. Securing network content. Technical report, PARC, Oct. 2009.

[86] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *LNCS*. Springer-Verlag, Berlin Heidelberg, 2005.

[87] R. Stewart. Stream Control Transmission Protocol. RFC 4960, IETF, September 2007.

[88] Randall Stewart, Michael Tuexen, Salvatore Loreto, and Robin Seggelmann. Stream Schedulers and a New Data Chunk for the Stream Control Transmission Protocol. Internet-Draft – work in progress 01, IETF, July 2014.

[89] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[90] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[91] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks (Fifth Edition)*. Prentice Hall, Upper Saddle River, NJ, USA, 2011.

[92] Michael Tuexen, Randall Stewart, Randell Jesup, and Salvatore Loreto. DTLS Encapsulation of SCTP Packets. Internet-Draft – work in progress 06, IETF, November 2014.

[93] Justin Uberti, Cullen Jennings, and Eric Rescorla. Javascript Session Establishment Protocol. Internet-Draft – work in progress 08, IETF, October 2014.

[94] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A Survey of DHT Security Techniques. *ACM Comput. Surv.*, 43(2):8:1–8:49, February 2011.

[95] Guillaume Urvoy-Keller and Ernst Biersack. A Congestion Control Model for Multicast Overlay Networks and its Performance. In *Proc. of 4th International Workshop on Networked Group Communication (NGC'02)*, 2002.

[96] Anne van Kesteren, Julian Aubourg, Jungkee Song, and Hallvord R. M. Steen. XMLHttpRequest Level 1. W3C Working Draft, World Wide Web Consortium, 2014.

[97] Christian Vogt, Max Jonas Werner, and Thomas 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, Oct. 2013. IEEEPress.

[98] Christian Vogt, Max Jonas Werner, and Thomas 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, Oct. 2013. IEEEPress. ICNP Best Demo Award.

[99] M. Waehlisch, T. Schmidt, and S. Venaas. A Common API for Transparent Hybrid Multicast. RFC 7046, IETF, December 2013.

[100] Matthias Wählisch, Thomas C. Schmidt, and Markus Vahlenkamp. Backscatter from the Data Plane – Threats to Stability and Security in Information-Centric Network Infrastructure. *Computer Networks*, 57(16):3192–3206, Nov. 2013.

[101] Matthias Wählisch, Thomas C. Schmidt, and Georg Wittenburg. BIDIR-SAM: Large-Scale Content Distribution in Structured Overlay Networks. In Mohamed Younis and Chun Tung Chou, editors, *Proc. of the 34th IEEE Conference on Local Computer Networks (LCN)*, pages 372–375, Piscataway, NJ, USA, October 2009. IEEE Press.

[102] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. RFC 1075, IETF, November 1988.

[103] Max Jonas Werner and Christian Vogt. Implementation of a Browser-based P2P Network using WebRTC. Technical report, Hamburg University of Applied Sciences, January 2014.

[104] Max Jonas Werner, Christian Vogt, and Thomas C. Schmidt. Let Our Browsers Socialize: Building User-centric Content Communities on WebRTC. In *Proc. of 34th Int. Conf. Dist. Comp. Systems ICDCS – WS HotPost*, pages 37–44, Piscataway, NJ, USA, June 2014. IEEEPress.

[105] G. Xylomenos, C. Ververidis, V. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. Katsaros, and G. Polyzos. A Survey of Information-Centric Networking Research. *Communications Surveys Tutorials, IEEE*, PP(99):1–26, 2013.

[106] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: defending against sybil attacks via social networks. *SIGCOMM Computer Communication Review*, 36:267–278, August 2006.

[107] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[108] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In Jason Nieh and Henning Schulzrinne, editors, *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '01)*, pages 11–20, New York, NY, USA, 2001. ACM.

Glossary

ALM Application Layer Multicast. 2, 85–87, 91–94, 96, 98–100, 102, 104, 107

ARQ Automatic Repeat-Request. 116

ASM Any Source Multicast. 88, 89, 95, 98

BOPlish Browser-based Open Publishing. 1, 22, 26

BOPlish ID Used as persistent identifiers for users.. 37

CDN Content Delivery Network. 18

Data Channel WebRTC transport that allows point-to-point transfer of generic data between Web browsers. 15, 17, 18, 36, 115, 119–121

DHT Distributed Hash Table. 5, 6, 66

DTLS Datagram Transport Layer Security [71]. 18

HTML Hypertext Markup Language. 9

HTTP Hypertext Transfer Protocol. 9

ICE Interactive Connectivity Establishment [75]. 13, 16, 18

ICN Information-centric Networking. 3, 18

IGMP Internet Group Management Protocol. 87–89, 91, 98

MLD Multicast Listener Discovery. 89, 91

Peer ID Volatile random numeric identifiers used to address BOPlish Peers. 37

RPF Reverse Path Forwarding. 20, 90, 92

SCTP Stream Control Transmission Protocol [87]. 17, 18, 115, 119, 120

SDP Session Description Protocol [41]. 15, 16

SOP Same-origin Policy. 75

SRTP Secure Real-time Transport Protocol [5]. 17

SSM Source Specific Multicast. 88, 89, 95, 98

STUN Session Traversal Utilities for NAT [76]. 16, 18

TURN Traversal Using Relays around NAT [66]. 16, 18

URI Uniform Resource Identifier. 9

User Community A group of users in BOPlish sharing a common interest. 22, 23, 28–32, 58, 60, 62, 66–68, 85, 108–110

WebRTC Web Real-Time Communication [4]. 11–13, 15–17, 115, 120

Hiermit versichern wir, dass wir die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt haben.

Hamburg, 12 December 2014 _____

Hamburg, 12 December 2014 _____