Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Implementation and Evaluation of a DHT-based content distribution system using WebRTC

Max Jonas Werner, Christian Vogt

Research Report

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Max Jonas Werner, Christian Vogt

Research Report

Implementation and Evaluation of a
DHT-based content distribution system using WebRTC submitted in the context of Projekt 2

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

Submitted on: October 7, 2014

Contents

1 Introduction 1

2 Background and Related Work 3
- 2.1 Publish/Subscribe Networking . . . 3
- 2.2 User-centric Naming and Networking . . . 4
- 2.3 WebRTC . . . 6
- 2.4 Previous Work and Contribution . . . 7

3 Conceptional Background 8
- 3.1 Data Transport Topology . . . 9
- 3.2 Message Format and URI Scheme . . . 9
- 3.3 ID Assignment . . . 11
- 3.4 Bootstrap Procedure . . . 11
- 3.5 Name Resolution and Data Routing . . . 14
- 3.6 Software Architecture Overview . . . 15

4 DHT Implementation 17
- 4.1 Software Overview . . . 18
- 4.2 Application Programming Interface . . . 20
- 4.3 Chord bootstrapping . . . 21

5 Emulation Environment 23
- 5.1 Headless Runtime . . . 24
- 5.2 Emulation Host . . . 25
- 5.3 Emulation Mediator . . . 26

6 Evaluation 30
- 6.1 Configuration . . . 31
- 6.2 Results . . . 31

7 Conclusions 37

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 our previous work, we introduced 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 prevents common privacy issues present in centralized architectures.

In this project report, we describe the implementation of a name resolution mechanism used to resolve location-independent names based on the BOPlish URI scheme. Moreover, we improved the user-facing API to allow 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 an emulation component capable of emulating a BOPlish system of a typical size.

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.

In the remainder, we give an overview of related work (Sec. 2). Sec. 3 showcases the concept of the BOPlish architecture. We describe the DHT-based routing layer in Sec. 4 and the emulation environment in Sec. 5. We follow up with an evaluation in Sec. 6 before concluding in Sec. 7.

2 Background and Related Work

2.1 Publish/Subscribe 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 [1]. In recent years, (proprietary) Content Delivery Networks 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 [2].

The ideas of Information-centric Networking (ICN) [3] have taken up the well-established concept of in-network storage and replication towards end-user communities, while adding the core objective of an open future Internet design. The latter requires resolution of the three major challenges *naming, security,* and *routing* [4]. 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 DONA [5] and NDN [6]. These and further solutions differ in naming, security, and routing, but all show a high interdependency among these three [7].

Unlike the Web URL-scheme, ICN uses names that are independent of a location or server instance. Two competing approaches exist, hierarchical and flat (e.g., hashed) names. ICN implementations like DONA and NDN use flavours derived from either of the two approaches (Figure 2.1).

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

Figure 2.1: 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, it is preferable to reduce

routing table sizes 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 unaggregatable identifiers requires either huge routing tables or external infrastructure. 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 [7].

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 [8].

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 some resolution service (e.g., DONA). In a coupled approach, data forwarding follows a path identified by the name resolution (e.g., a source route). In a decoupled approach, data is forwarded independently of content routing paths using regular IP/BGP routing [3].

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 [4].

2.2 User-centric Naming and Networking

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 [9], 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 hostname 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 [9] 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 [10]. 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 [11]. 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 [12] that is based on RELOAD specified in [13]. 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 [12]. 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.

2.3 WebRTC

WebRTC is a protocol suite that enables two Web browsers to communicate directly over a UDP-channel [14], paired with a JavaScript API for Web applications [15]. WebRTC transfers A/V data via the Secure Real-time Transport Protocol (SRTP) as well as generic binary and textual data via the Stream Control Transmission Protocol (SCTP) over Datagram Transport Layer Security (DTLS). Because most browsers are expected to operate behind a NAT, the Interactive Connectivity Establishment protocol (ICE) is natively provided. ICE uses the Session Traversal Utilities for NAT (STUN) and its extension Traversal Using Relay NAT (TURN) to circumvent connectivity problems in NAT environments. Fig. 2.2 shows a stacked view of the protocol suite. WebRTC is limited in the way that it allows two browsers to interconnect and exchange data. The standards neither include topology- nor routing-related topics.

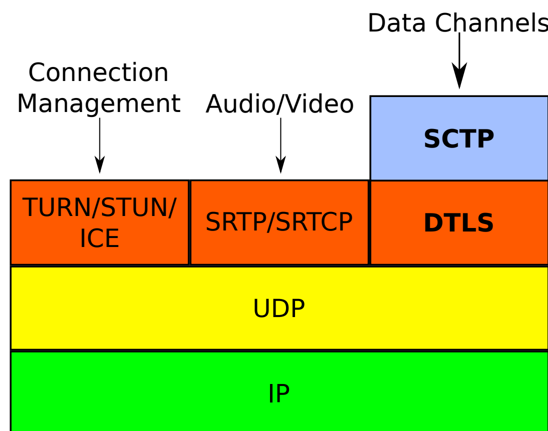

Figure 2.2: WebRTC Protocol Stack

Current research based on the WebRTC technology is mostly conducted in the multimedia conferencing and CDN context. The authors of [16] present a media server component that exploits the expected broad deployment of WebRTC to converge multimedia conferencing on different devices like smartphones and desktops. They provide an architecture based on open source software that handles media mixing, transcoding and filtering for group communication use cases.

Maygh [17] is a WebRTC-based system that facilitates P2P content distribution among participating clients. Maygh uses a centralized P2P lookup system with a coordinator node to store mappings between content and clients that already downloaded specific content.

Succeeding requests from other clients can than be answered by peers that already downloaded the content with the help of the coordinator.

We build our architecture for application networking on the emerging WebRTC standards, which are under active implementation in several browsers. The technology is or is expected to be broadly available on a large number of devices and therefore acts as an enabler for our system.

2.4 Previous Work and Contribution

We described a first implementation of our Browser-based Open Publishing (BOPlish) approach in [18]. The paper laid out the core software architecture and described applications building on top of the framework. While being fully functional, the P2P routing layer maintained a full mesh between all participating peers thus limiting scalability. We further refined our concept in [19] by introducing a naming scheme based on URIs. We validated our concept by investigating if and how specific use cases can be implemented.

The work at hand adds the following contributions. We extend the architecture of the system and exchange the full mesh routing mechanism with a DHT-based one. The DHT allows the system to scale without sacrificing the absence of centralized infrastructure and adds the functionality of a distributed key/value store. We identify emulation support as a crucial component to prove our conceptual assumptions and therefore introduce an emulation environment to conduct measurements of a larger scale. Moreover, we evaluate the system using realistic numbers of peers to prove our conceptional assumptions.

3 Conceptional Background

The solution presented here focuses on the idea that users have complete control over naming and providing content and eventually introduces a user-centric naming and networking concept, similar to those presented in [9], [10] or [11].

A user community consists of a number of peers that are connected to each other via a P2P network. A Web server delivering a BOPlish application serves as bootstrap 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 address) and the user has to authenticate at an identity provider. The combination of peer id and username is then used to join the user community and stored in a Distributed Hash Table (DHT) with the username as key and the peer id as value.

Our reference implementation of this concept consists of a JavaScript library that 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`. 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.

The first milestone of our implementation attempt enabled us to build P2P applications like a chat and a simple game [18]. We built a server component for bootstrapping each peer and connecting new peers to existing ones. On the client-side we had implemented a Connection Manager for abstracting the rather complex process of establishing new WebRTC connections in a P2P network as well as a Router component that forwarded messages to the correct peers.

In this project report we describe the next steps we performed in order to implement our vision of a generic user-centric, infrastructure-independent content-sharing facility. We

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

exchanged the full-mesh Router component with a Chord DHT implementation, added the functionality to handle BOPlish URIs and established a baseline for automatic testing of our code. On the signaling layer we defined the protocol and data types more specifically.

3.1 Data Transport Topology

BOPlish is comprised of two transport layers: a signaling layer used for connection establishment messages and a routing layer used for passing application-specific messages through the network to a designated peer. The messages on the signaling layer are exchanged either via WebSockets through the bootstrap server or via WebRTC Data Channels, depending on the availability of a Data Channel to the peer. E.g. when connecting to a community, the first message has to be sent through the rendezvous server. The messages on the routing layer are always exchanged via WebRTC Data Channels and sent through the network to 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 DHT, a bootstrap node has to be chosen to initialize the Chord joining procedure.

Thus, in the sections of this chapter we differentiate between a 'bootstrap server' and a 'bootstrap node', where the former is used to open a WebRTC data channel to the latter.

3.2 Message Format and URI Scheme

All messages exchanged in BOPlish are encoded in JSON. A message consists of `to` and `from` fields which denote the sender and receiver of a message. Moreover, the message contains a protocol-specific string identifier (`type` field) and a `payload` field for arbitrary data. The message format thus reads like this:

```
1 {
2   to: "<peer id receiver>",
3   from: "<peer id sender>",
4   type: "<type identifier>",
5   payload: {
```

```
6         <JSON-formatted payload>
7     }
8 }
```

When the user community is established, content can be published, accessed and shared among the peers. For this purpose, the design of content identifiers is a key ingredient to our solution. 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 [20]. 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 or e-mail. These are some examples of BOPlish URIs:

`bop:alice@example.org:document/image.png?sha-256;1234abc...`

`bop:bob@example.com:search/Music/*tomte*`

BOPlish URIs guarantee a location-independence by employing the username instead of a specific peer identifier. The actual peer id 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 ID Assignment

As stated above, we distinguish peer id and BOPlish id. The peer id is a unique but temporary and location-dependent identifier. The BOPlish id is a user-friendly unique, persistent and location-independent name used for addressing content. The peer id has three purposes [13]:

- To address the node itself.
- To determine the node's position in the DHT.
- To determine the data set for which the node is responsible.

Assigning identifiers to peers in a secure way is discussed in several publications ([21] 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. 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. With regard to our BOPlish concept any of the three mechanisms may be employed, 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.

3.4 Bootstrap Procedure

Every peer in a BOPlish user network first acquires a unique id (the peer id) using one of the mechanisms stated above and establishes a WebSocket connection to the chosen bootstrap server using that id. After the WebSocket connection to the bootstrap server has been established, a new peer sends a bootstrap message to the server (Fig. 3.1 shows the complete bootstrap process). The generation of that message is triggered by the API call `ConnectionManager.bootstrap()`. Such messages are denoted by the "to" field set to the value "*" and the "type" field set to "signaling-protocol". The payload contains the offer generated by

the WebRTC API call `PeerConnection.createOffer()`[2]. 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>"
  }
}
```

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 [18]). When joining an existing network (with at least one peer), the initial offer is passed by 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.

The chosen peer, after receiving the initial offer, answers with a signaling message of type "answer" and the corresponding answer SDP. This answer is also sent through the bootstrap server. Upon receiving this answer, the two peers have established a Data Channel connection.

From this point on, the established 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. 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.

[2] http://dev.w3.org/2011/webrtc/editor/webrtc.html#widl-RTCPeerConnection-createOffer-void-RTCSessionDescriptionCallback-successCallback-RTCPeerConnectionErrorCallback-failureCallback-RTCOfferOptions-options

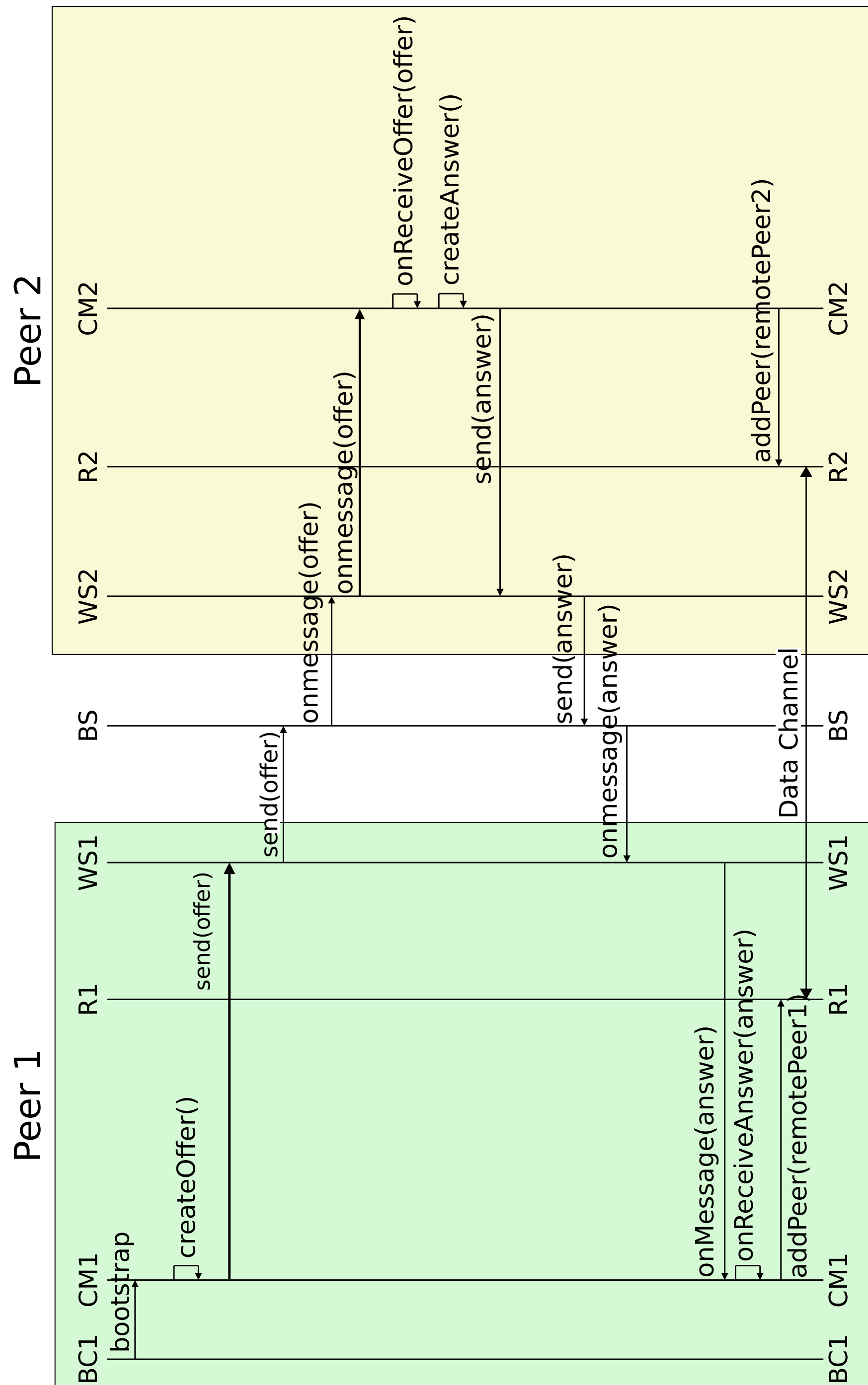

Figure 3.1: Sequence diagram of the bootstrap process in BOPlish. Peer 1 generates an offer, sends it through a WebSocket connection to the bootstrap server which then selects a candidate used for bootstrapping, in this case, peer 2. Then, peer 2 generates an answer, sends it back through the bootstrap server, eventually resulting in a Data Channel between the two peers.

3.5 Name Resolution and Data Routing

Our concept of user-centric content networks revolves around the idea that every participant in a specific P2P browser network 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.

A mechanism is needed to unbind the relation between a current location and the content identifier. ICN features such a mechanism but operates on the network layer and therefore requires deep changes to the network infrastructure. BOPlish introduces an overlay that does not depend on external infrastructure but is formed solely from the participating peers. The 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 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.

DHTs tend to be fragile when peers join/leave the network in a high frequency [22]. 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 uses only a sparsely filled data structure to prevent re-organizing from having a big impact on the system. This is achieved by only storing the identifier-location linkage, not the content itself. As a result, the DHT can be designed to be highly churn-resistant and redundant.

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 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.

The reference to a location is obtained by using the return value of the DHT name resolution procedure. 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.

3.6 Software Architecture Overview

The design of our architecture is presented in Figure 3.2. 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.

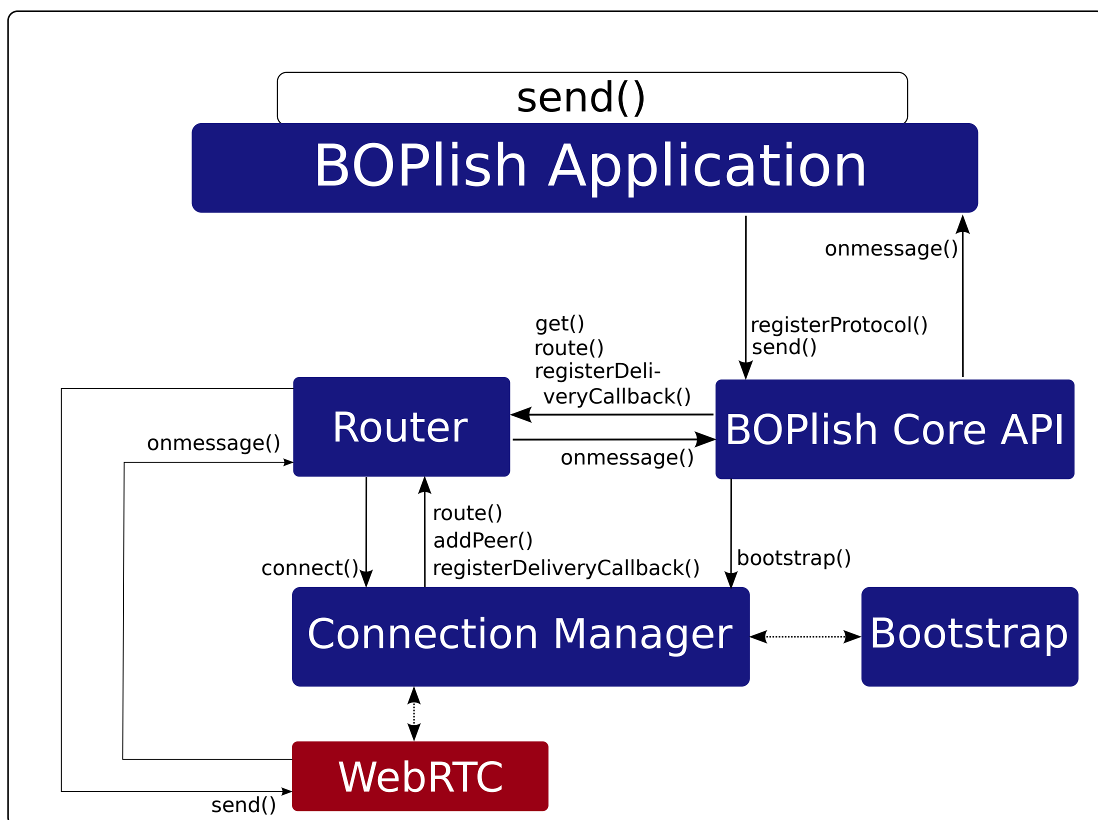

Figure 3.2: Overview of the BOPlish software architecture

The Router component is responsible for deciding where to forward messages to and thus maintains a routing table, eventually forming a peer in the DHT. It exposes a KBR API as defined in [23] that hides the DHT implementation introduced in Sec. 4. 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. The Router encapsulates messages into the routing format (see Sec. 3.2) 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 DHT Implementation

As described in Sec. 3, we use a Distributed Hash Table (DHT) as a name resolving mechanism. In this chapter, we provide a detailed look into our DHT implementation. We opted for the Chord protocol [24] for the following reasons. Compared to other DHT protocols, the Chord paper provides detailed implementation information and even pseudo-code; additionally, Chord has proved to be working in large-scale implementations [13]. Exchanging Chord with another protocol is unproblematic due to the standardized KBR API that BOPlish uses. The KBR API exposes a set of well-defined program calls and acts as a middle layer between various P2P protocols and the application itself.

BOPlish differs from typical implementations due to the underlying Browser environment. All existing Chord implementations we know about (such as the widely known OpenChord[1]) use a kind of socket-based API or Remote Procedure Calls (RPC). Those implementations are based on TCP or UDP sockets for communication. 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,

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

which is a constraint imposed by current Browsers) and stores peer ids instead of IP/port combinations.

- Contrary to other Chord implementations, all communication between BOPlish peers is asynchronous due to nature of the Data Channel interface. This avoids the common problem of head-of-line blocking and enables the Router to process multiple requests simultaneously.

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.

4.1 Software Overview

As pictured in Figure 4.1, 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 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.

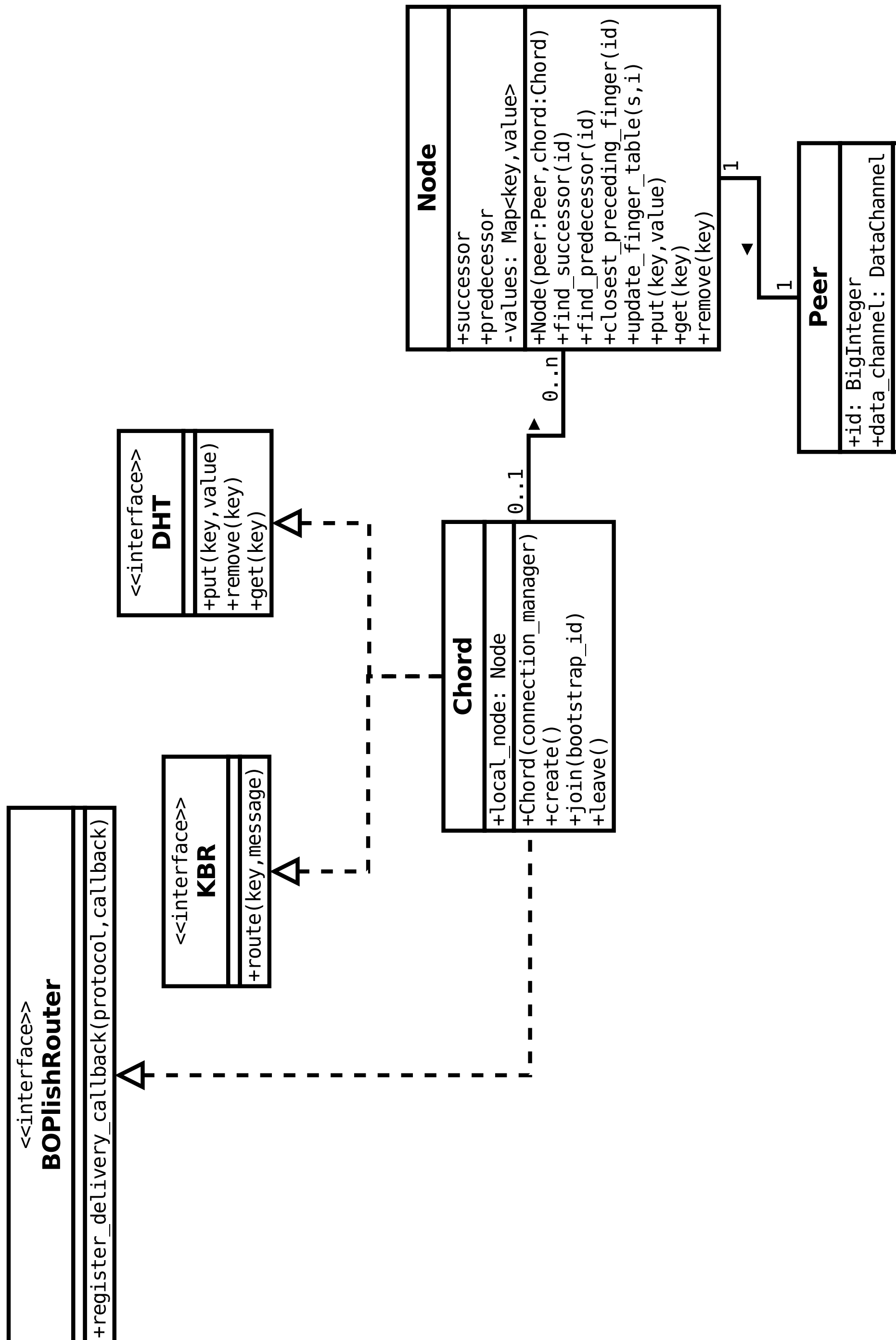

Figure 4.1: Class diagram of the BOPlish Chord implementation

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. Thus, we implemented a way to map responses that arrive through the Data Channel to the requests sent earlier. To solve this problem we introduced a mechanism that uses sequence numbers for marking transactions (consisting of a request and a response). 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 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. 6 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.

4.2 Application Programming Interface

As shown in the class diagram of Figure 4.1 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 [23]. 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 [18], where it is called `setOnMessageHandler()`. A sample usage of the DHT API is illustrated in the following code snippet:

```
var chord = new Chord();
chord.join(42, function(err) {
    if(err) {
        throw new Error("Error joining");
    }
    chord.put(12345, {
            name: "Hans Blix",
            profession: "politician"
        },
        function(err) {
            chord.get(54321, function(obj) {
                console.log(obj);
            });
    });
});
```

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 stdout.

4.3 Chord bootstrapping

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

1. Initialize new node’s finger table
2. Update existing node’s finger tables
3. Copy key/value pairs to new node

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

```
{
  to: "<peer id receiver>",
  from: "<peer id sender>",
  type: "chord-protocol",
  payload: {
      seqnr: 0,
      type: "FIND_SUCCESSOR",
      id: "<peer id sender>"
  }
}
```

The type “chord-protocol” 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-protocol” 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. 4.1. The “id” field contains the ID of the first finger table entry’s start.

The bootstrap node eventually answers with a message of the following form, providing the requester with a data structure containing the successor’s node ID as well its successor and predecessor nodes’ ID:

```
{
  to: "<peer id receiver>",
  from: "<peer id sender>",
  type: "chord-protocol",
  payload: {
      seqnr: 0,
      type: "SUCCESSOR",
      successor: {
          id: "<successor ID>"
          successor: "<successor's successor ID>"
          predecessor: "<successor's predecessor ID>"
      }
  }
}
```

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 [24].

5 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 initally built for unit-testing purposes but also serves as a basic building block for the emulation component.

The actual emulation environment can be seperated 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. 5.1 shows the system architecture.

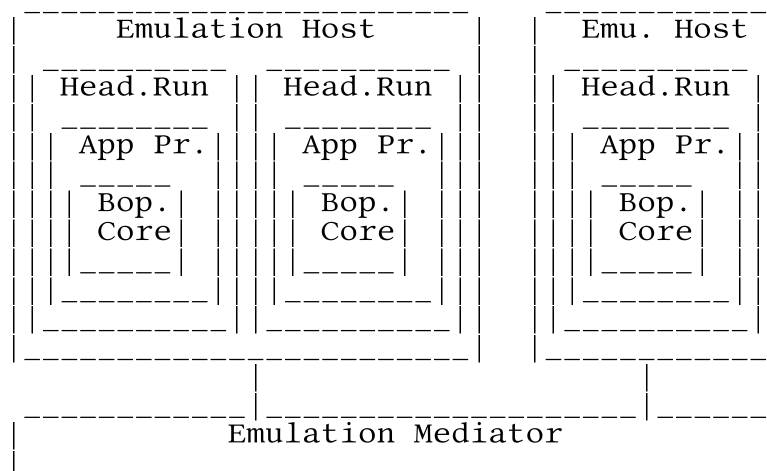

Figure 5.1: Emulation Environment Architecture

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).

5.1 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 occuring 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 seperated into two categories: *browser automation frameworks* and *headless JavaScript runtimes*. Solutions like Selenium[1] 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 abilites 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[2] 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 test cases. Mocha[3] is a widely adopted example of such a framework. Tests are written in JavaScript code and executed by using a Node.js[4]-based test runner. This enables running

[1] http://seleniumhq.org/
[2] http://nodejs.org/
[3] http://visionmedia.github.io/mocha/
[4] http://nodejs.org/

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[5]. 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.

5.2 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. 5.2.

#	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

Figure 5.2: 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 `id`, while call 4 returns a list of all the peers currently running on this host. The returned `ids` can then be used to stop the peers or gather status information. It is also possible to abort

[5] https://github.com/js-platform/node-webrtc

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. 5.3, can therefore gather all messages in a central place. During the WebSocket initialization, a filter can be specified to prevent overloading from happening.

5.3 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[6] or the frontend interface described below.

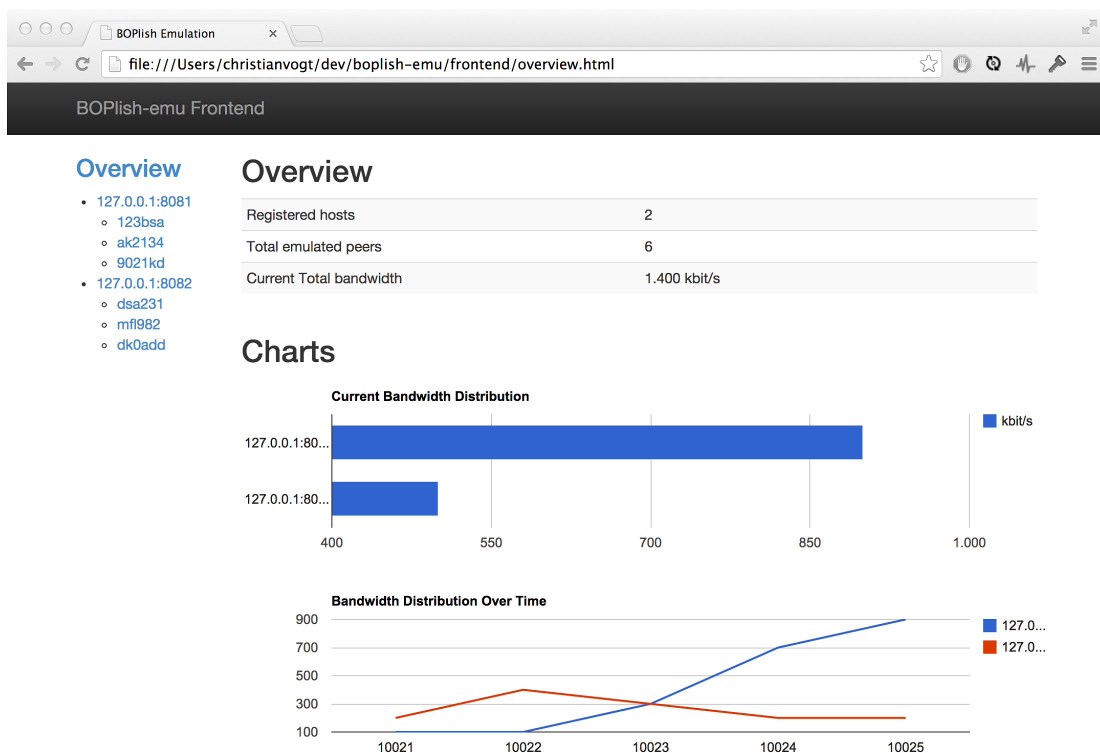

Figure 5.3: Emulation Overview Page

[6]http://curl.haxx.se/

The frontend interface shown in Fig. 5.3 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. 5.2.

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 ge 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. 5.3) gives a summary of the current overall emulation status. All the gathered data is available to the chart engine[7] 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 gatherd log files. The data available to the host overview page include:

- Current bandwidth/sec of all running hosts
- 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. 5.4. 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
- Host uptime

[7] based on Google Charts (`https://developers.google.com/chart`)

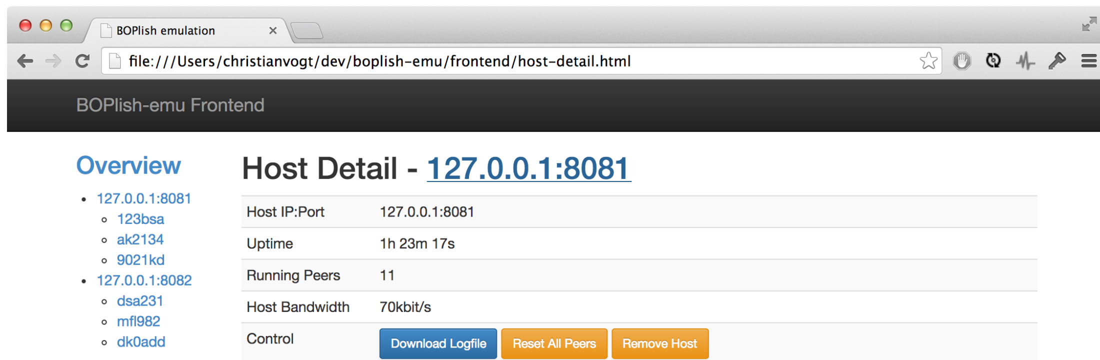

Figure 5.4: Emulation Host Detail Page

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 occuring 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 absolut order. For that task, NTP can be used to reduce the time difference to a acceptable level (lower than the hop-by-hop delay).

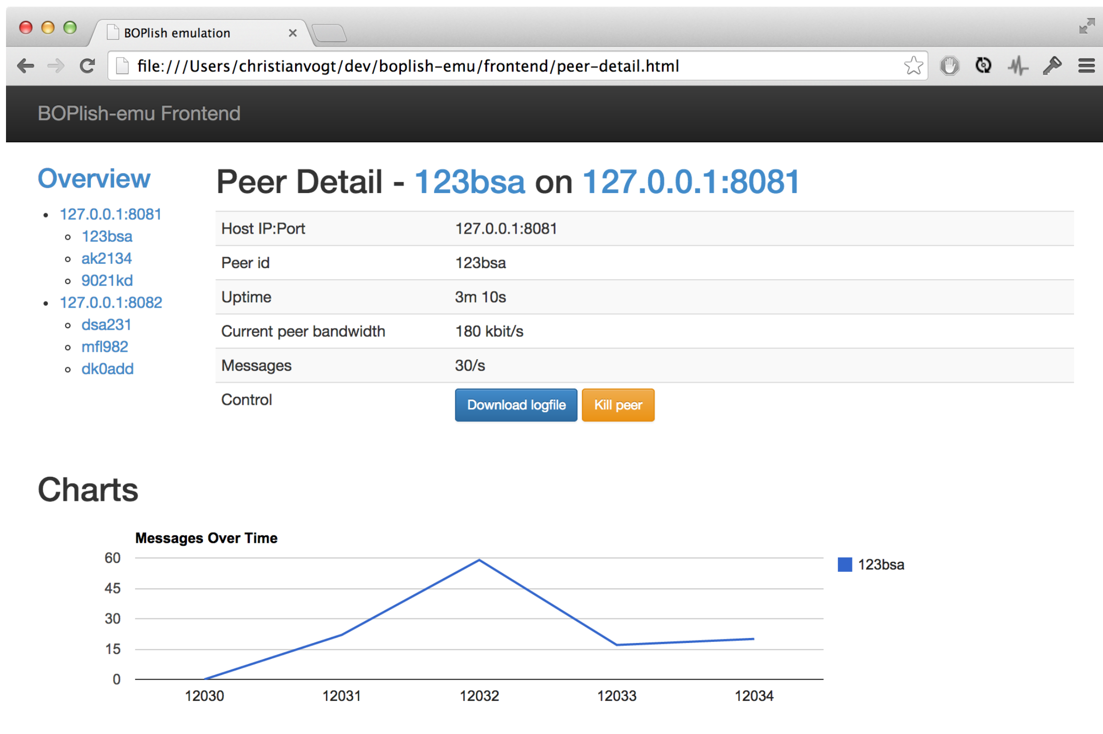

Figure 5.5: Emulation Peer Detail Page

6 Evaluation

After laying out the principles of our architecture in Sec. 3 and Sec. 4, we now continue to test the system for its functionality using the emulation component introduced in Sec. 5. BOPlish uses a DHT mechanism for its name resolution capabilities that is formed solely from the participating peers. The underlying DHT protocol used to establish and maintain the DHT depends on the use case of the BOPlish community.

Currently, the 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 to set options for the SCTP stream that is used for the Data Channel connections. As such, only reliable transmission can be tested. Another limiting factor is the number of open Data Channels a peer 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). For this reason, we do not give an detailed insight on performance characteristics at this point but rather test the system for functionality and conceptional correctness.

For small-sized communities, One-Hop DHTs are feasible. Every peer knows every other peer in the community and can therefore determine the owner of a key in a single step. Lookup performance is obviously very good ($\mathcal{O}(1)$) as no intermediate hops are involved. On the other hand, a lot of maintenance traffic is needed to keep the host tables updated at every peer in the event of churn. Moreover, as stated above, browsers can only handle a limited number of open Data Channel connections.

Larger communities will need to use a multi-hop DHT setup. We implemented the Chord protocol for our test environment but other protocols like Pastry or CAN are certainly possible, too. The logarithmic scalability of Chord allows for large numbers of peers. In our tests, we compare One-Hop (as occurring in One-Hop DHTs) as well as Two-Hop and 10-Hop performance. First up, we provide a look at the system configuration in Sec. 6.1 and showcase the results in Sec. 6.2.

6.1 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. This turned out to be a problem with the STUN implementation of the Chrome Browser as it apparently expects to operate in a NATed environment. 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 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 reflect the Internet like PlanetLab[2].

We currently do not use any caching for the name resolution mechanism. Thus, every request to a BOPlish id gets resolved to a corresponding peer id prior to the actual content transmission. We plan to enable lookup caching after the functionality has been verified.

6.2 Results

Lookup Performance

The first test is the lookup performance that resembles the name resolution mechanism. We issue GET requests to the DHT layer using the following code:

```
1  var counter = 100;
2  var getDelays = [];
3  var hostBopId = 'bwjllzqiitpgb@id.com';
4  (function getDelay() {
5    var timeStart=new Date();
6    bopclient._get(hostBopId, function(err, msg) {
7      var timeDiff = new Date() - timeStart;
8      console.log('took', timeDiff, 'ms');
9      getDelays.push(timeDiff);
10     if (--counter) {
```

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

```
      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);
}
```

Due to the low-delay LAN environment, the measured delays almost solely mirror the actual delay introduced by the BOPlish overlay and the WebRTC stack. Fig. 6.1 shows the results of our tests. One-hop performance shows a minimum of $4ms$ and a maximum of $43ms$. As expected, two-hop delays are roughly doubled. 10-hop delays show a big span between minimal and maximum values. This result might seem unexpected in a stable LAN environment but can be explained by the heavy-weight WebRTC stack. We expect the overall delays to decrease when WebRTC implementations mature over time.

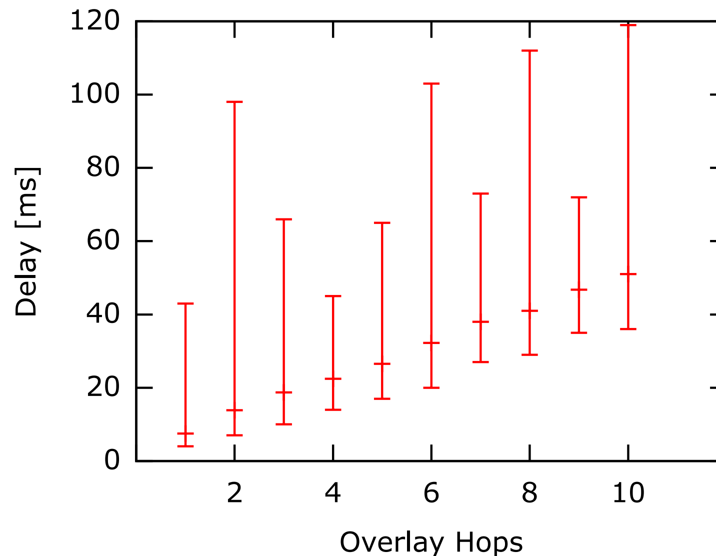

Figure 6.1: BOPlish lookup performance

Bootstrap Delay

To offer the best 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.

Fig. 6.2 outlines the gross bootstrap delay in the environment described above and indicates the total time from instantiating the BOPlishClient object until the peer has fully 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. The chart can roughly be divided into two parts: The first part is the one where only two peers have joined the DHT, the second part is the one displaying the delay with three and more hops on the X axis.

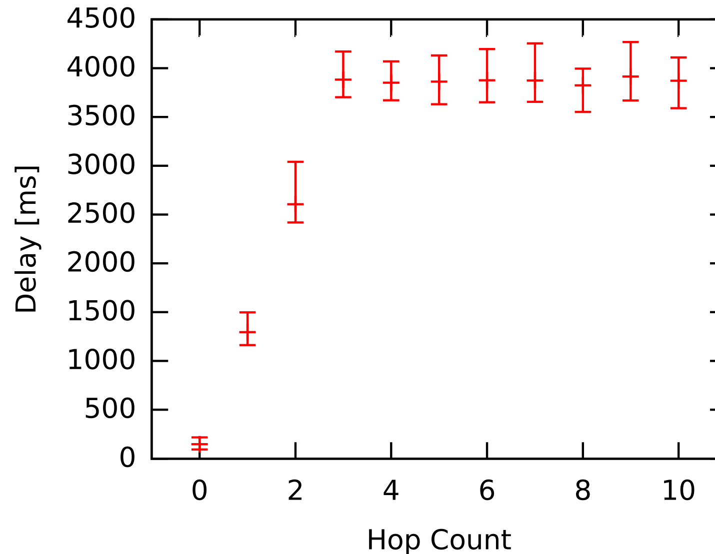

Figure 6.2: Gross BOPlish bootstrapping performance

It can be seen that the maximum delay for bootstrapping remains more or less constant with increasing hop count. This has two reasons:

1. When joining, a peer connects to at most three other peers: its bootstrap peer, its successor and its predecessor.

2. The delay for establishing a WebRTC Data Channel is very high.

Those two aspects combined allow for a reasonable interpretation of Fig. 6.2. The connection establishment delay is so high that all other operations on the resulting Data Channels are negligible with regards to delay measurements. The conclusion from this observation is that Data Channel connections are expensive to establish and it's important to keep as many of them open as possible, at least to those peers that are contacted often (like successor and predecessor).

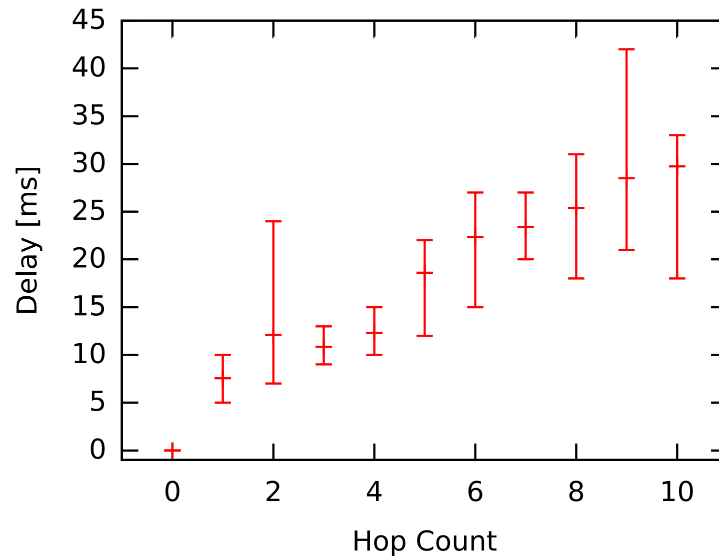

Figure 6.3: Net BOPlish bootstrapping performance

We also measured the net bootstrap delay, leaving out the time for connection establishment. Those numbers provide for a better evaluation of our actual implementation. Fig. 6.3 displays the time from instantiating the BOPlishClient until the join operation has succeeded as a function of the hop count used to route the join messages. Here it can be seen that the bootstrap delay increases with more peers joining the DHT, this increasing the hop count for join messages.

DHT Performance Characteristics

To test the overall stability of the DHT, we conducted the maximum number of message per second that can be routed over an increasing hop count. We learned that the results differ depending on the WebRTC implementation respectively the browser we used in our tests. The following code has been used to conduct the measurements:

```
1  var hostBopId = 'drnlbbnzs@id.com';
2  var start = new Date();
3  var j = i = 1000;
4  while (i--) { // send i messages to bop id
5    bopclient._get(hostBopId, function(err, msg) {
6      if (!--j) {
7        // done: all callbacks called
8        var took = new Date() - start;
9        // normalize to msg/sec
10       var msgPerSec = 1000*1000/took;
11       console.log('msg/sec:', msgPerSec);
12     }
13   });
14 }
```

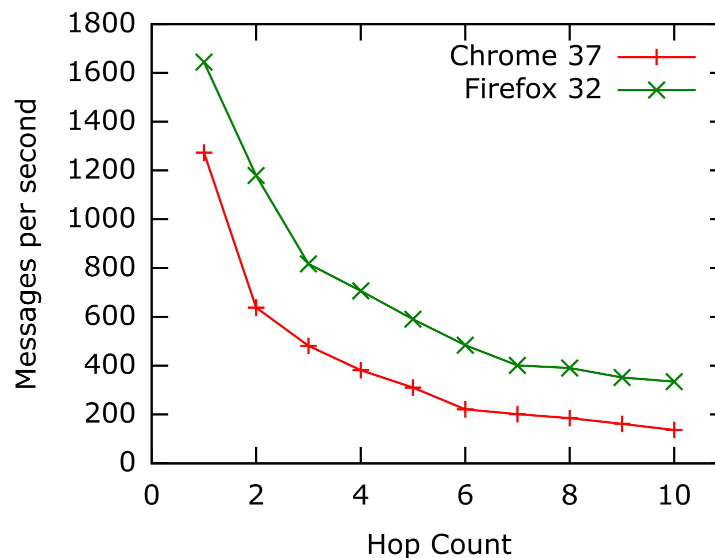

Figure 6.4: BOPlish DHT Performance

The results are shown in Fig. 6.4. 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 is able to push $1640 \frac{msg}{sec}$ through the BOPlish infrastructure. 10-hop performance is more than doubled compared to Chrome with $330 \frac{msg}{sec}$. The results leave us confident that our solution can sustain even high amounts of messages and does not break down in case of overloading.

Loss

We did not observe any packet loss during the tests at all. This is due to the reliable transmission mode of the underlying SCTP connection and the LAN environment.

7 Conclusions

In this paper, we introduced two independent extensions to the BOPlish architecture, namely the DHT extension for the name resolution mechanism and the emulation environment. Accompanied by the implementation of the URI scheme, BOPlish can now scale to a realistic community size needed for the implementation of the introduced use cases.

In the process of our implementation efforts we experienced various pitfalls with current WebRTC implementations: The API, developed by the W3C in collaboration with the IETF is still under heavy development and changes are introduced very often with new runtime releases (e.g. Firefox or Chrome). This drastically reduces the implementation pace of WebRTC-based applications since debugging errors with new browser releases is a time-consuming task. Our conclusion from various such situations are that writing unit and integration tests are not only important but constitute a critical building block in developing WebRTC applications. Another learning, especially with regards to our emulation component, is that the currently available WebRTC stacks demand vast CPU and RAM resources. It is not clear whether this situation will change in the near future since currently, implementors put more effort in the specification and development of the API.

We conducted several measurements to showcase that the principle architecture of BOPlish works as expected. During the evaluation, we discovered major differences in the performance of current WebRTC implementations. As these implementations are currently considered work-in-progress we expect to see future performance improvements.

Our BOPlish implementation is now in a state where we are able to put effort in the further development of the conceptional framework. We are currently planning to design a group communication mechanism that makes use of our name-based publishing architecture.

Bibliography

[1] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, June 2003.

[2] M. Handley, "Why the Internet Only Just Works," *BT Technology Journal*, vol. 24, no. 3, pp. 119–129, Jul. 2006.

[3] 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*, vol. PP, no. 99, pp. 1–26, 2013.

[4] D. Kutscher, S. Eum, K. Pentikousis, I. Psaras, D. Corujo, D. Saucez, T. C. Schmidt, and M. Wählisch, "ICN Research Challenges," IRTF, IRTF Internet Draft – work in progress 02, February 2014. [Online]. Available: http://tools.ietf.org/html/draft-kutscher-icnrg-challenges

[5] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A Data-Oriented (and beyond) Network Architecture," *SIGCOMM Computer Communications Review*, vol. 37, no. 4, pp. 181–192, 2007.

[6] V. Jacobson, D. K. Smetters, J. D. Thornton, and M. F. Plass, "Networking Named Content," in *Proc. of the 5th Int. Conf. on emerging Networking EXperiments and Technologies (ACM CoNEXT'09).* New York, NY, USA: ACM, Dec. 2009, pp. 1–12.

[7] A. Ghodsi, T. Koponen, J. Rajahalme, P. Sarolahti, and S. Shenker, "Naming in Content-oriented Architectures," in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, ser. ICN '11. New York, NY, USA: ACM, 2011, pp. 1–6.

[8] D. Smetters and V. Jacobson, "Securing network content," PARC, Tech. Rep., Oct. 2009.

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

[10] T. Callahan, M. Allman, M. Rabinovich, and O. Bell, "On Grappling with Meta-information in the Internet," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 5, pp. 13–23, Oct. 2011.

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

[12] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, H. Schulzrinne, and T. Schmidt, "A SIP Usage for RELOAD," IETF, Internet-Draft – work in progress 13, July 2014.

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

[14] H. Alvestrand, "Overview: Real Time Protocols for Browser-based Applications," IETF, Internet-Draft – work in progress 11, August 2014.

[15] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "WebRTC 1.0: Real-time Communication Between Browsers," http://www.w3.org/TR/2013/WD-webrtc-20130910/, World Wide Web Consortium, W3C Working Draft, 2013.

[16] L. Lopez Fernandez, M. Paris Diaz, R. Benitez Mejias, F. Lopez, and J. Santos, "Kurento: a media server technology for convergent WWW/mobile real-time multimedia communications supporting WebRTC," in *Proc. of 14th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'13)*, June 2013, pp. 1–6.

[17] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram, "Maygh: Building a CDN from Client Web Browsers," in *Proc. of 8th ACM European Conference on Computer Systems (EuroSys'13)*. New York, NY, USA: ACM, 2013, pp. 281–294.

[18] M. J. Werner and C. Vogt, "Implementation of a Browser-based P2P Network using WebRTC," Hamburg University of Applied Sciences, Technical Report, January 2014. [Online]. Available: http://inet.cpt.haw-hamburg.de/teaching/ws-2013-14/master-project/Prj1-report-werner-vogt.pdf

[19] M. J. Werner, C. Vogt, and T. 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.* Piscataway, NJ, USA: IEEEPress, June 2014.

[20] M. Wählisch, T. C. Schmidt, and S. Venaas, "A Common API for Transparent Hybrid Multicast," RFC Editor, RFC 7046, December 2013. [Online]. Available: http://tools.ietf.org/html/rfc7046

[21] G. Urdaneta, G. Pierre, and M. V. Steen, "A Survey of DHT Security Techniques," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 8:1–8:49, Feb. 2011.

[22] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling Churn in a DHT," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

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

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