WebDHT: browser-compatible distributed hash table for decentralized Web applications

Lorenzo Rossi
Department of Physics, Informatics and Mathematics
University of Modena and Reggio Emilia
Modena, Italy
lorenzo@rossilorenzo.dev

Luca Ferretti
Department of Physics, Informatics and Mathematics
University of Modena and Reggio Emilia
Modena, Italy
luca.ferretti@unimore.it

***Abstract*—Modern browser technologies allow running highly portable and usable complex applications. However, the inability to access all the operating system features may limit their features or performance when compared to native software in certain scenarios. We investigate the design of peer-to-peer (P2P) networks of interconnected browsers to improve applications interconnecting users, such as videotelephony, messaging and gaming. Although peer-to-peer protocols are well-established in the literature, known designs and implementations cannot be executed on browsers due to constraints of browser environments. We propose *WebDHT*, a webassembly library for creating P2P networks among browsers which offers topic-based peer-discovery features and integrates usable identity authentication mechanisms. WebDHT implements a variant of the Kademlia protocol based on distributed hash tables (DHT) adapted to support WebRTC protocol. WebDHT requires a native server to be available only for network bootstrap, but leverages existing browsers connected to the DHT to decentralize WebRTC signaling backends. We propose an open-source implementation and two demonstrative applications for users messaging and multimedia streaming, and analyze limitations and future work for designing better browser-compatible P2P networks.**

***Index Terms*—WebRTC, DHT, Kademlia**

I. Introduction

A large number of modern Internet applications are implemented as *browser-compatible* applications, that is, they are written through browser technologies (JavaScript and WebAssembly-compliant programming languages) and are compliant with browsers APIs. Advantages over native software include cross-compatibility over multiple platforms and operating systems, no installation procedures, easier integration with the Web ecosystem, and improved security thanks to browsers sandboxes. However, their inability to access all the operating system features may also limit their performance and capabilities in certain scenarios.

We focus on applications which enable communications among users, such as videotelephony, messaging and gaming applications, and we investigate the possibility of designing a browser-compatible application where user browsers communicate with each others in a decentralized fashion through a peer-to-peer network. Historically, Web protocols have been based on centralized paradigms where browsers can exchange data among each other through centralized relay servers through HTTP and WebSockets. The more recent WebRTC protocol, which is mainly known for low-latency multimedia communications, allows browsers to establish peer-to-peer connections to directly exchange arbitrary data. However, establishing WebRTC connections still requires adopting signaling channels which are typically deployed through centralized servers, thus falling back to a centralized design.

We propose WebDHT, a browser-compatible library that creates a peer-to-peer network of browser nodes, enabling decentralized peer discovery and peer-to-peer communications while minimizing reliance on centralized servers, sharing operations load and data on all nodes. WebDHT is designed with scalability and usability as core objectives, and is meant to be used as a signaling layer by multiple Web applications. By sharing signaling cost, applications using WebDHT can minimize deployment costs while maximizing service uptime. WebDHT can be considered as a variant of the Kademlia protocol [1]: it is based on Distributed Hash Tables (DHT) but modifies the original protocol to support browser technologies and Web protocols, including support of connection-oriented protocols instead of datagram-oriented protocols due to reliance on WebRTC to establish peer-to-peer communications. Although peer-to-peer networks are considered quite established in the literature [1] and many mature implementations exist [2], [3], none of them consider constraints related to the execution within browser environments.

We propose a prototype implementation of WebDHT which can be used both as a high-performance server and as a WebAssembly-compliant library written in RUST that offers high performance and exposes a JavaScript API. The design of WebDHT is modular over its transport protocol, allowing easier testing and debugging. The project has a well-defined high-load behavior, is designed to be resistant to attack vectors that are typical of peer-to-peer networks and to overcome many technological issues related to current browsers. We also propose two applications to showcase how a real-world Javascript Web application can use WebDHT to discover peers. The first is a simple room-based chat, the second is a screen-sharing and live multimedia sharing application, useful to synchronize media playback in a group of users.

The remainder of the paper is as follows. Section II describes base knowledge. Section III and IV present the high-level and detailed designs of WebDHT. Section V describes technical details of the implementation. Section VI proposes a discussion related to technologies, performance and security. Section VII compares the proposal with related work.

II. Base knowledge

A. Kademlia DHT

A Distributed Hash Table (DHT) can be modeled as a hash table deployed within a distributed network. It allows each node to associate *values* to *keys*, and to retrieve them efficiently given the *key*. Values are bitstrings of arbitrary size and each key is a n-bit bitstring, where n is a security parameter. We denote the key space as $\mathbb{K} := \{0,1\}^n$. On average, key-values entries are distributed uniformly among all the nodes of the network. Each node is identified by $\mathtt{ID} \in \mathbb{K}$ and maintains a set of entries associated to keys that are *closer* to its identifier with regard to the identities of the other connected nodes. The distance is computed through a metric which is specific for each DHT protocol. Kademlia [1] uses the XOR operation as a distance metric. To add a new value in association to a key, a node must query the DHT network to obtain the set of node identifiers that are closer to the key, and ask them to store the value. The number of nodes that will be asked to store the value is a system-wise parameter. Any node can retrieve nodes that are closer to a target ID and the associated addresses to contact them. To this aim, each node maintain a so-called *k-bucket* routing table which can be modeled as a tree over the space of the identifiers, and stores more information regarding identifiers that are closer to its own. In Kademlia, each node $\hat{\mathtt{ID}}$ maintains an array of n buckets of size k, and each bucket $i \in [1,\ldots,n]$ includes at most k identifiers ID that are at a distance $d = (\hat{\mathtt{ID}} \oplus \mathtt{ID})$ such that $2^{i-1} \leq d < 2^i$. Thus, each bucket i includes all the identifiers that share i leading bits, and a larger number of identifiers must be discarded for greater values of i. Although the original Kademlia paper [1] proposes a dynamic strategy which adapt the sizes of the buckets at runtime, popular Kademlia implementations adopt a simplified static approach which we also adopt in this paper.

B. WebRTC

WebRTC is a protocol for low-latency audio/video communications with a focus on Web browsers. It is very popular for interactive communications among users and is becoming increasingly adopted also for real-time streaming. It supports both direct peer-to-peer communications between browsers and communications relayed through intermediate servers, but also supports data-only channels which are typically used for communicating auxiliary information. Its protocol stack includes the Real Time Protocol (RTP) based on UDP, and supports tuning of packet loss for better performance-reliability trade-offs. Figure 1 shows how a peer-to-peer WebRTC connection channel is opened through a pre-existing *signaling channel*. The initiator of the connection, namely the *caller*, must send an *offer* message through the signaling channel to a recipient, namely the *callee*, which must respond with an *answer* message. Offer and answer messages include information to negotiate the new channel parameters, such as IP addresses, and UDP and TCP ports, encoded through the *Session Description Protocol (SDP)*. The establishment of the channel is operated by using the *Interactive Connectivity Establishment (ICE)* protocol to handle devices potentially separated by Network Address Translation (NAT) mechanisms. The protocol establishes secure communication channel by using self-generated x509 certificates, which cannot be accessed by browser APIs except for their fingerprint. We denote as *signaling backend* the mechanism used to implement the signaling channel, which is outside the scope of the standard. Typical signaling backends include communications relayed by a centralized Web server through HTTP or Web Sockets. One of the most important design choices of WebDHT is to minimize reliance on centralized servers by allowing existing WebRTC connections among nodes to act as signaling backends by leveraging existing WebRTC channels.

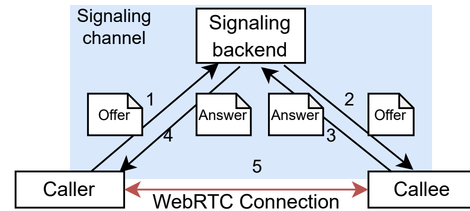

Fig. 1. WebRTC handshake via an abstract signaling channel

C. WebAssembly

WebAssembly is a standard binary instruction format for executable programs released in 2017 to allow execution of very efficient programs on browsers. Native programming languages like C and Rust can be compiled to WebAssembly, allowing re-use on multiple platforms. WebAssembly programs are also able to access features of the native operating system through the Web Assembly System Interface (WASI). The proposed prototype implementation of the WebDHT protocol is implemented in Rust, a memory-safe strongly-typed native language that has official support for WebAssembly. This decision has been taken to share most of the code between a browser client and an optimizable native server.

III. WebDHT Design Overview

WebDHT builds a network of interconnected peer-to-peer nodes, where each node can be either a *native node* or a *browser node*. Native nodes execute as native programs on a machine operating system. Browser nodes execute in a browser environment and are more restricted in terms of allowed operations, available APIs, and other constraints related to browser technologies and Web protocols. Any node can connect to a browser node only through a common neighbor, which can be either a native or a browser node. At bootstrap, a browser node can interact only with native nodes with the HTTP protocol. Later, the browser node can connect to another browser node with the WebRTC protocol either via a browser or a native node, which is used as signaling channel. WebDHT guarantees high fault-tolerance because even in case of some neighbors failures, nodes can continue operating through other neighbors thanks to information distributed to many nodes of the DHT. We design WebDHT as a middleware library to be used by applications to manage data within the DHT and to open

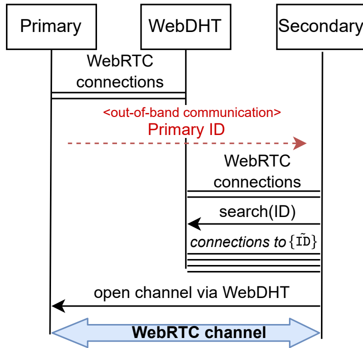

Fig. 2. Example of WebDHT communication with an out-of-band communication channel to distribute nodes identifiers

synchronous peer-to-peer communications with other nodes connected to the network. To these aims, each node exposes peer interfaces that can only be executed by other nodes after establishing a mutually authenticated communication channel, which guarantees protection against identity spoofing attacks and consistency of operations on the DHT.

An application that uses WebDHT can use an out-of-band channel to distribute its identifier and allow other nodes to open new connections. An example related to a primary-secondary communication paradigm is shown in Figure 2. To ease peer discovery by applications, WebDHT also offers a topic-based identity discovery mechanism tightly integrated within the DHT. While identifiers are generated through a probabilistic procedure (that is, the generation of the asymmetric key pair), topics can be chosen arbitrarily. Thus, they can be easily used by applications to implement user-friendly discovery procedures (e.g., people communicating a topic name via short messages) or to integrate peer discovery with applications communication contexts (e.g., the label identifying a chat room).

WebDHT adopts WebRTC to establish peer-to-peer communications among nodes, introducing multiple challenges with regard to known peer-to-peer networks specifications.

Connection-oriented WebRTC communications. Modern peer-to-peer networks based on DHTs typically rely on datagram-oriented communication paradigms because they do not need expensive features of connection-oriented communications. However, we are constrained by browser technologies, and must comply to the connection-oriented paradigm of WebRTC communications, which also requires intermediate signaling channels.

Decentralization of signaling channels. WebDHT does not leverage dedicated centralized signaling backends to handle WebRTC signaling channel (see Section II-B). Instead, WebDHT allows any neighbor node to act as an intermediate signaling node to establish new WebRTC connections. We call these intermediate nodes as *referral nodes*. A native node can act as a referral node for itself by offering an HTTP endpoint. However, WebDHT must implement signaling channels on connections opened between browser nodes, thus adopting established WebRTC connections as signaling channels for new WebRTC connections. This logic must be implemented through browser technologies.

Main design traits that distinguish WebDHT from existing DHTs include mechanisms to efficiently handle connections life-cycle, taking into account that the maximum number of active WebRTC connections on a browser is much lower than that supported by a native operating system, and that even by knowing the address of a browser node, opening a connection first requires opening intermediate connections to common neighbors in order to form a path, WebRTC connections are used both for routing mechanisms and as transport protocols for applications. To this aim, the WebDHT library implements a hybrid approach for integrating connection pooling and routing mechanisms that maintains both a minimal amount of stable connections to efficiently operate routing mechanisms within the decentralized network, while still allowing to establish ephemeral connections for applications features. Additional designs features include avoiding potential double connections that may be established due to delays, handling connections pooling to possibly re-use previously opened connections, and cryptographic channel-bindings bind WebRTC and DHT authentication mechanisms.

IV. WebDHT Protocol Details

We describe notations in Section IV-A, setup operations in Section IV-B, connection establishment in Section IV-C, node peer interfaces in Section IV-D, library interfaces in Section IV-E, topic-based operations in Section IV-F.

A. Parameters and notations

We denote the DHT key space as $\mathbb{K} := \{0,1\}^n$, where n denotes a security parameter of the system, a node identity as $\mathtt{ID} \in \mathbb{K}$, a (cryptographic) collision-resistant hash function as $\mathcal{H}_n(\cdot) : \{0,1\}^* \to \{0,1\}^n$ (e.g., a truncated SHA-2 or SHA-3 function). We model the local hash table of each node as an associative array $M := \{\mathtt{K} : \{\mathtt{ID} : \mathtt{V}\}\}$ that maps each key $\mathtt{K} \in \mathbb{K}$ to an other associative array that maps identities $\mathtt{ID}$ to values $\mathtt{V} \in \{0,1\}^*$, where $\mathtt{ID}$ tracks the node that published value $\mathtt{V}$. We denote the local routing table of each node as $R := \{\mathtt{ID} : \mathtt{conn}\}$, which maps each identifier $\mathtt{ID}$ to an open connection $\mathtt{conn}$ maintained by the WebDHT library. As in Kademlia, given a target node $\mathtt{ID}$, the routing table efficiently returns the top-t $\mathtt{IDs}$ that are closer to $\hat{\mathtt{ID}}$, and stores different amounts of entries depending on the distance between stored identities and the node identity (see Section II-A). However, our routing table distinguishes from that designed by Kademlia because R associates $\mathtt{IDs}$ with open connections instead of addresses. If an $\mathtt{ID}$ is stored within the routing table, then there is always an open connection with the node identified by $\mathtt{ID}$. If a connection is closed, then the associated entry is removed. Moreover, the routing table stores a limited amount of entries and old entries may be discarded, thus involving that old connections may be forcefully closed if they are not currently used by the library.

B. Setup

Node setup refers to initialization operations of a node before connecting to the DHT. The node generates an asymmetric

key pair (sk, pk) and derives its ID from the public key pk as $\texttt{ID} \leftarrow \mathcal{H}_n(pk)$.

Network setup refers to operations for creating and running a new WebDHT network. First, it is required to setup one native node. If a public ICE server is not available, one or multiple dedicated ICE servers may be deployed as well for enabling NAT traversal features of WebRTC (see Section II-B). For improved resiliency, other native nodes may be added to the DHT through a *node bootstrap* procedure (see Section IV-E). At least an address of a connected native node and of an ICE server must be distributed to nodes that need to participate in the WebDHT network.

C. WebRTC connection establishment

Contacting browser nodes within WebDHT requires establishing a WebRTC connection, which involves a *caller* node that sends an *offer* through the signaling channel to a *callee* node, which responds with an *answer*. *Offer* and *answer* messages comply with the *SDP* specification, and include connection-related information such as endpoint candidates. For details on WebRTC see Section II-B.

The procedures required to establish a WebRTC connection depends on the type of *callee* node and on the information available to *caller* nodes.

- the first scenario involves a *caller* that is either a native or browser node, that must connect to a *callee* that is a native node, by knowing its IP address (or hostname);
- the second scenario involves a *caller* that is either a native or browser node, that must connect to a *callee* that is either a native or browser node, by only knowing its identity ID within the WebDHT network.

We observe that traditional DHTs only involves the first scenario, because all nodes can typically accept data from other nodes by exposing an endpoint (e.g., a UDP listening socket), and are known by their address(es).

In the first scenario, the *callee* can act as a referral for itself. Indeed, native nodes can offer an HTTP endpoint as a signaling channel to create a new WebRTC connection. Both native and browser nodes can create an *offer* and send it through an HTTP request to the native node. The request is processed and an *answer* is returned as the body of the HTTP response. Depending on security requirements, HTTPs can be used as well without relevant modifications.

In the second scenario, the *callee* and the *caller* nodes must have a common neighbor node (either native or browser) which can act as a referral node. In this case, establishing a new WebRTC connection to the *callee* node requires a proper protocol to forward WebRTC signaling information and to authenticate nodes. We describe this protocol by referring to Figure 3, where a *caller* Node A connects to a *callee* Node C via a common *referral* Node B. The protocol requires that Node A knows the identity of Node C $\texttt{ID}_C$, and that Node B is a common neighbor of Node A and Node C, that is, there exist established connections between Node A and Node B, and between Node C and Node B.

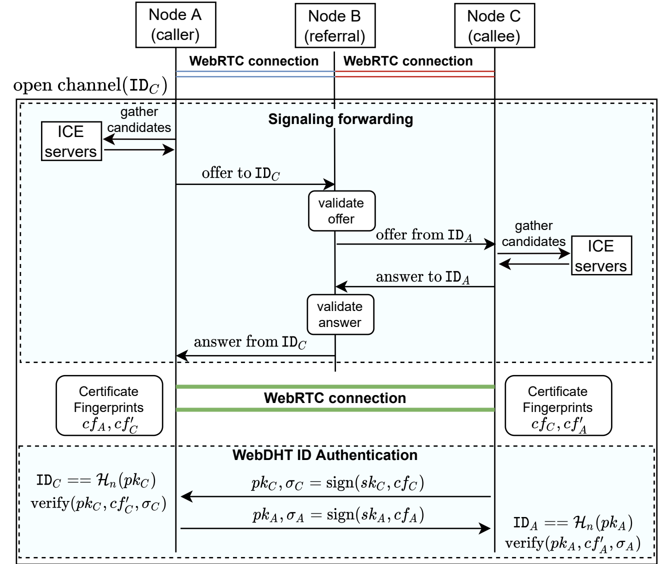

Fig. 3. WebDHT protocol for connecting to a node by knowing its ID

The first phase of the protocol involves *signaling forwarding*. Node A creates a WebRTC offer with the collaboration of known ICE servers, and asks Node B to forward the offer to Node C. Node C receives the offer and interacts with known ICE servers to produce an answer for Node A, which is sent via Node B. To prevent misuse, this request-forwarding protocol is only permitted for signaling. As an example, Node B does not forward any other message from Node A to Node C, and thus Node A must establish a direct connection to send any other data or request to Node B.

After the first phase, Node A and Node C have established a direct WebRTC connection. Thus, they have exchanged public certificates used to secure the connection. Note that as for the WebRTC standard (see Section II-B), these certificates have been automatically generated by browsers and it is not possible to use a PKI to authenticate them. Moreover, the WebDHT library executed within browser environments can only access their fingerprints. To establish the authenticity of the certificates, Node A and Node C operate an additional handshake procedure, denoted as *WebDHT ID Authentication*, which uses the nodes identities as trust anchors. Within the handshake, Node A knows identity $\texttt{ID}_C$ from the input of the open channel procedure, and Node C knows identity $\texttt{ID}_A$ from the offer received within the *signaling forwarding* phase. We denote as cf_A and cf_C the actual fingerprints of the certificates (locally accessed by Node A and Node C, respectively), and cf'_A and cf'_C as the fingerprints of the certificates received by Node C and Node A, respectively. If the WebRTC connection has been established securely (e.g., there is no ongoing man-in-the-middle attack), cf_A (cf_C) and cf'_A (cf'_C) are equal. The authentication procedure proceeds independently for Node A and Node B, which must operate analogous operations. The procedure operated by each node can be considered as a challenge-response protocol to prove possession of the WebDHT private key generated at setup time (see Section IV-B), and uses the certificate fingerprint

as a challenge to verify that the endpoints of the WebRTC connections are indeed Node A and Node C. For simplicity, we only describe authentication of Node C at Node A. Node C signs its certificate fingerprint by using its WebDHT private key as $\sigma_C = \texttt{sign}(sk_C, cf_C)$ and sends it to Node A together with its WebDHT public key pk_C. Node A verifies the authenticity of pk_C by comparing it to the known identity of Node C, as $\texttt{ID}_C == \mathcal{H}_n(pk_C)$. Then, it verifies the authenticity of the known certificate fingerprint cf'_C against the received signature as $\texttt{verify}(pk_C, cf'_C, \sigma_C)$.

Managing double connections. A potential issue of using WebRTC as a transport protocol for a DHT is that when two peers want to communicate with each other they may open communication channels almost simultaneously, possibly opening two WebRTC connections (*double connections*) and thus wasting resources. As the number of open connections in browser environments is very limited, a WebDHT implementation let each peer to only accept the connection initiated by the peer with the lower ID. This ensures that only one connection between the peers is opened. To this aim, both peers must check this condition and prevent having two connections between each node.

D. Peer operations for connected nodes

Each node offers four RPC-like operations for nodes that have previously connected via a valid transport protocol. In the following, we denote as $\bar{\texttt{ID}}$ the identity of the caller node that executes an operation, which is an information that any operation can obtain from the connection context (see Section IV-C above).

Routes retrieval $\tilde{R} \leftarrow \texttt{prget}(\texttt{ID}, t)$ returns a subset of the node routing table $\tilde{R} = \{\texttt{ID}\}$ which includes identifiers of known nodes that are closer to the given identifier ID within the local routing table R, where t denotes the maximum size of the set. If the node does not store any closer node, it returns an empty set.

Value retrieval $(\{\texttt{V}\}, \{\tilde{\texttt{ID}}\}) \leftarrow \texttt{pvget}(\texttt{K}, t)$ returns the values $\{\texttt{V}\}$ associated to a given key K stored within the local hash table M, or the empty set if K is not within M. Moreover, it executes $\texttt{proutes}(\texttt{K}, t)$ to return nodes with closer identifiers $\tilde{\texttt{ID}}$ within R.

Value insertion $\texttt{pvinsert}(\texttt{K}, \texttt{V}, \texttt{TTL})$ inserts a key-value entry $(\texttt{K}, \texttt{V})$ in the local hash table M for a maximum expiration time equal to an input time-to-live (TTL). That is, it stores a new entry $\bar{\texttt{ID}} : \texttt{V}$ associated to K within M. The node behaves with a best-effort approach, as it will store the value in the DHT if enough space remains, and may discard the value before the expiration of the TTL if needed.

Value deletion $\{\tilde{\texttt{ID}}\} \leftarrow \texttt{pvdelete}(\texttt{K}, t)$ deletes the entry associated to key K and identifier $\bar{\texttt{ID}}$, if exists. A node $\bar{\texttt{ID}}$ can only delete values inserted in the DHT by itself. The operation returns the set of identifiers $\{\tilde{\texttt{ID}}\}$ of known nodes that are closer to the given key K within the local routing table R, where t denotes the maximum size of the set. If the node does not store any closer node, it returns an empty set.

E. Library operations

We consider operations that can be executed by an application running on a browser or native node which uses the WebDHT library. We assume that all operations can access to key pair (sk, pk) and node identity $\bar{\texttt{ID}}$ generated at setup time (see Section IV-B), and can query ICE servers when necessary.

Node bootstrap $\texttt{wdht} \leftarrow \texttt{bootstrap}(\texttt{addresses})$ allows an initialized node to join the WebDHT network. The node must know one or multiple addresses associated with available native nodes and ICE servers. The node interacts with any of the available native nodes as follows. The node generates n *dummy* identities chosen at random within each bucket of its routing table (let $\bar{\texttt{ID}}$ denote the node identity, the node generates $\{\texttt{ID}_i\}_{i\in[1,\ldots,n]}$ such that $2^{i-1} \leq (\bar{\texttt{ID}} \oplus \texttt{ID}_i) < 2^i$). The node searches for identifiers closer to each $\texttt{ID}_i$ on the DHT via a $\texttt{wdht.nsearch}(\texttt{ID}_i)$ (see below). The operation inserts all nodes queried throughout the execution of the operation within the node routing table R. Moreover, all intermediates nodes also acquire routing information about the caller node $\bar{\texttt{ID}}$ when establishing connections. This procedure ensures a probabilistic lookup complexity of $\mathcal{O}(\log N)$ queries for following lookups.

DHT nodes search $\tilde{R} \leftarrow \texttt{wdht.nsearch}(\hat{\texttt{ID}}, t)$ returns identities of the connected nodes $\tilde{R} = \{\tilde{\texttt{ID}}\}$ that are closer to $\hat{\texttt{ID}}$ within the DHT. The operation uses an iterative approach. In the first iteration, the node that executes the operation retrieves up to t nodes $\tilde{\texttt{ID}}$ that are closer to $\hat{\texttt{ID}}$ included in its routing table R, establishes a new connection with each of the nodes and queries each of them to get up to further t closer nodes as $\texttt{conn}_{\tilde{\texttt{ID}}}.\texttt{prget}(\hat{\texttt{ID}}, t)$. At the second iteration, the returned nodes that are closer to the target $\hat{\texttt{ID}}$ are also queried, and so on for the following iterations. The procedure ends when the target identity $\hat{\texttt{ID}}$ is found or when queried nodes do not return nodes that are closer to $\hat{\texttt{ID}}$ with regard to the previous iteration. If $\hat{\texttt{ID}}$ exists, a connection with it is established as well and inserted in the routing table.

DHT value search $\{\texttt{V}\} \leftarrow \texttt{wdht.vsearch}(\texttt{K}, t)$ returns values stored within the DHT associated to an input key K. Its implementation can be considered as a variant of *DHT node search*, but queried nodes that store values associated to K return them to the requester. Given K, the operation looks for the nodes with identifiers $\{\tilde{\texttt{ID}}\}$ that are closer to K with an iterative approach by leveraging the $\texttt{conn}_{\tilde{\texttt{ID}}}.\texttt{pvget}(\texttt{K}, t)$ peer operation. If the queried nodes return nodes with identities $\tilde{\texttt{ID}}$ that are closer to the target K, they are queried on their turn. Distinct values $\{\texttt{V}\}$ potentially returned at each operation are locally accumulated to be returned to the application. The operation terminates as the *node search* operation, when queried nodes do not return any closer node.

DHT value insertion $\texttt{wdht.insert}(\texttt{K}, \texttt{V}, \texttt{TTL})$ inserts a key-value $(\texttt{K}, \texttt{V})$ entry within the DHT associated with a time-to-live (TTL). Its execution involves the execution of *node search* with input K as $\{\tilde{\texttt{ID}}\} \leftarrow \texttt{nsearch}(\texttt{K}, t)$. Each node $\tilde{\texttt{ID}}$ is requested to insert the entry with the *value insertion* peer operation $\texttt{conn}_{\tilde{\texttt{ID}}}.\texttt{pvinsert}(\texttt{K}, \texttt{V}, \texttt{TTL})$. The value of t

determines the redundancy of the entry within the DHT.

DHT value deletion $\texttt{wdht.delete}(\texttt{K}, t)$ deletes the entry associated with key K and identifier $\tilde{\texttt{ID}}$ within the DHT. It represents a variant of *DHT node search*, where queries to the nodes invoke the *value deletion* peer operation as $\{\tilde{\texttt{ID}}\} \leftarrow \texttt{pvdelete}(\texttt{K}, t)$, which deletes the given entry if it exists within the hash table M of the queried node, and returns closer nodes $\{\tilde{\texttt{ID}}\}$ to be queried on their turn. The procedure terminates when queried nodes do no return closer nodes. The value of t may influence the number of iterations within the procedure.

F. Topic-based identity advertise and discovery

Operations for identity advertise and discovery based on topics are derived from primitive library operations described in the previous Section IV-E, as follows.

DHT topic advertise $\texttt{wdht.advertise}(\texttt{T}, \texttt{meta}, \texttt{TTL})$ advertises the identity of the node that executes the operation (ID) in association with topic T. The operation is implemented by inserting identity ID and metadata meta as values associated to a key $\texttt{K}_\texttt{T} \leftarrow \mathcal{H}(\texttt{T})$, that is by executing $\texttt{wdht.insert}(\texttt{K}_\texttt{T}, \texttt{meta}, \texttt{TTL})$).

DHT topic discovery $\{\tilde{\texttt{ID}} : \texttt{meta}\} \leftarrow \texttt{wdht.discover}(\texttt{T})$ return identities associated with topic T by retrieving values associated to a key $\texttt{K}_\texttt{T} \leftarrow \mathcal{H}(\texttt{T})$ within the DHT by executing $\{\texttt{ID} : \texttt{V}\} \leftarrow \texttt{wdht.vsearch}(\texttt{K}_\texttt{T})$ and by decoding each value V as meta).

We observe that, when using a human-friendly string as topic, it is advised to use a hierarchical naming convention to support context-separation. As an example, an app named *appname* created by *example.com* might use *com.example.appname.discovery.v1* to prevent conflicts with other applications that use the same WebDHT instance. Moreover, since WebDHT adopts a best-effort approach to establish the persistence of each entry within the DHT, periodic re-publishing may be performed by applications even before the given TTL. An example communication flow based on topic advertise and discovery is shown in Figure 4. The example shows a primary-secondary communication paradigm where two nodes are able to establish a communication channel by using a pre-shared topic, possibly known by users. The example can be easily extended to other communication paradigms among multiple nodes (e.g., a mesh network) by letting the application discover all nodes associated to a topic and establish a communication channel with each of them.

V. WEBDHT IMPLEMENTATION

We provide an implementation of WebDHT based on the Rust programming language which supports both WebAssembly and native programs[1]. We leverage async features of Rust to offer a high-performance native node, but partially disable them when compiling in WebAssembly environments due to missing support for threads. As we show in Figure 5, the project has been organized into multiple modules to satisfy the following characteristics: to be able to compile different portions of the source depending on the target environment (browser or native nodes); to support multiple transport protocols (WebRTC and HTTP); to allow efficient testing for complex logic, potentially in simulated environment. The project is organized in *core* modules, *adapters* and *applications* that we describe in the following.

[1] https://github.com/SnowyCoder/wdht/tree/nca2022

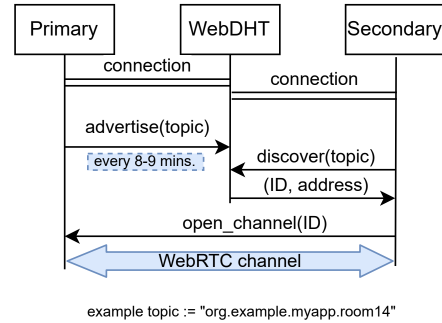

Fig. 4. Example of WebDHT communication with identity advertise and discovery based on shared topics

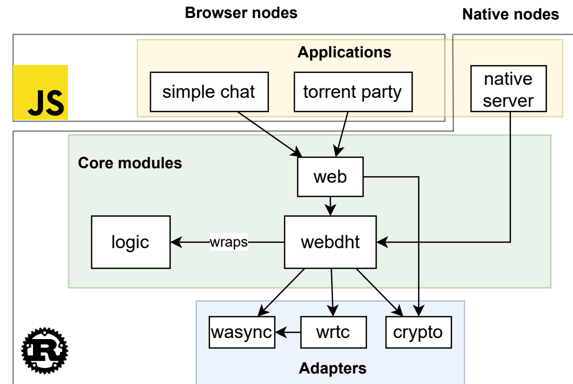

Fig. 5. WebDHT implementation organization

The core modules of the project are *logic*, *webdht* and *web*. The *logic* module defines the DHT logic, implementing the routing table and the storage layer. This layer is independent of the transport protocols, but implements a lightweight dummy transport layer that can be used for local simulations. The *webdht* module provides a working WebDHT and can be directly used by Rust projects. It wraps the *logic* module to add a WebRTC transport layer and an HTTP client for bootstrap operations. It also provides an HTTP endpoint that can be used in native nodes for WebRTC signaling. The *web* module provides a user-friendly API accessible from JavaScript code in a browser environment. This module can be exported as a WebAssembly NPM module.

The *adapter* modules allow conditional compilation depending on the target environment without affecting the complexity of the code. The *crypto* module exports abstractions for cryptographic operations to offer the best performance in both native and browser environments. For browser targets, the module wraps the WebCrypto API, achieving better performance than WebAssembly implementations and reducing the size of the

binary distributed over the Web. For native targets, the module uses the popular p256 [4] and sha2 [5] libraries. The *wasync* module provides abstractions over asynchronous functions to allow spawning of new tasks and async timeout. The *wrtc* module provides a simplified WebRTC API. For browser targets, it uses the standard Web APIs for WebRTC [6]. For native targets, it uses the popular libdatachannel [7] library. To the best of our knowledge, a WebAssembly-compatible WebRTC client library is, to this day, missing in the open-source ecosystem. Thus, we consider that this module could be of independent interest and could be used by external projects with few modifications.

Finally, our implementation includes *applications* for deploying a WebDHT network and for demonstration purposes. The *native server* module provides a standalone, configurable, high-performance minimal server implemented with Rust which can be used to deploy a native node. This is the only module that cannot be compiled into browser-compatible WebAssembly as it requires native capabilities. The *simple chat* module is a very simple Javascript application to demonstrate the capabilities of WebDHT. It implements a Web text chat for groups organized in rooms. The application uses WebDHT as a peer-discovery mechanism. When the application starts it allows to either join a random room or join a room by specifying its name. Rooms are mapped to topics within WebDHT for an easier access by people. Once joined, the application creates a mesh network of WebRTC connections among all the other members of the room. The *torrent party* module is a more complex Javascript application for synchronizing media among a group of people. It implements the same room management and mesh networking paradigm of the *simple chat*, but allows users to leverage two features for uploading a file or for screen sharing. The application leverages the *WebTorrent* library to share it with other people in the same room and, if possible, to stream as a HTML5 video element, synchronizing the playback time.

VI. Analyses and Discussions

A. Limits and issues related to browser technologies

WebDHT leverages WebRTC data channels, which are much less popular than multimedia channels typically used for audio/video calls. Thus, browser developers give lower priority to related bugs and issues. During the project development we found various bugs in every prominent browser engine. A major issue in Chromium-based browser is the very limited number of possible WebRTC connections. Currently, after 500 connections, Chromium-based browsers fail at creating new connections [8]. This puts a hard limit on the degree of a WebDHT browser node and might also deter multiple sites from using the project as the limit is instance-wise. Even if some connections get closed, the browser might not be able to open new connections due to another bug: WebRTC connections are not counted in garbage cleaning statistics. To re-use previously closed connections the applications need to wait a garbage-cleaning cycle or cause one manually by allocating and de-allocating a large quantity of memory. Firefox also has WebRTC-related issues where UDP candidates are sometimes not gathered on a page reload, as this bug is not easy to isolate nor reproduce we could not report the issue. Another issue common in both engines on various operating systems is a timeout before candidate gathering is complete [9], Chromium developers describe it as "intended behavior". WebDHT requires additional development to work around the issue.

Another more fundamental issue is caused by background page hibernation. In modern browsers when a page is not actively used it is put in a state of limited resources. In WebDHT this behavior can be observed as applications may be unable to answer to WebRTC requests. This behavior slows down significantly the project bootstrap and operations since a new node needs to wait a predefined timeout before dropping the connection. Potential solutions might include detecting page focus state, to notify peers if a hibernation event is occurring, or to escape page hibernation completely.

Future developments which may help the development of browser-based DHTs are web-push notifications [10], which could be used as a signaling channel or as a bootstrap option to reconnect to previous nodes. Moreover, W3C is working on DHT-related WebRTC use cases that may help the project's development like reusable SDPs and 1RTT connection establishment [11].

B. Behavior in case of high-loads

We implement WebDHT to support high load scenario even with limited resources. In case of browser nodes, the major constraint is related to the limited amount of connections available on browsers. In case of native nodes, the major bottleneck may be represented by those used to bootstrap the network, which may have to support a large number of requests. Kademlia-based DHT routing tables have limited capacity: if a k-bucket of the routing table is full, no other node is inserted. The node will still accept connections from other nodes and answer their queries, but their information will not be used for routing purposes. Our WebDHT implementation uses a similar approach to handle high-load scenarios with little variants. Every node has two main limits, one for the routing table and a higher one for the transport-level connections. When a connection to the node is established, it will try to insert it in the routing table. If the routing table is full, the new connection will be put in a list of "half-closed" connections, and the other peer is notified about this behavior. A new connection that caps the transport-level limit causes the server to close an other available half-closed connection, if possible. The new connection is rejected only if no connections can be closed. To enforce a fair behavior, our implementation adopts a First In First Out queue for half-closed connections.

C. Behavior in case of Sybil and Eclipse attacks

DHTs are inherently vulnerable to Sybil and Eclipse attacks. In a Sybil attack, the adversary creates many "fake nodes" within the DHT to gather information about the issued queries

or to disrupt routing with ill-responding nodes that do not contribute to the routing protocol. Instead, an eclipse attack aims at polluting the routing table of a target node until it only stores Sybil nodes. Both Sybil and Eclipse attacks leverage the self-management of identities by the nodes. Previous work on MainlineDHT [12], [13] and Emule's DHT [14] render less effective these attacks by letting nodes derive their identity from some publicly verifiable network or transport layer information, such as a public IPv4 or the port number. This approach cannot be used by WebDHT because WebRTC currently does not use UDP multiplexing and any new connection uses a new port. Moreover, deriving the identifier from IPv4 alone would not support clients behind NATs, which is a typical scenario for browser nodes. WebDHT instead derives its routing nodes identities only from the cryptography public key, without using any network or transport-related information. Thus, an adversary must randomly sample the identity-space by generating new keys, and cannot freely choose identifiers closer to a target value. Future work on WebDHT may include supporting the query protocol described in S/Kademlia [15] to render eclipse attacks unfeasible.

VII. Related work

The traditional approach to implement a signaling channel for WebRTC is based on centralized services, for which exist popular implementations. As an example, PeerJs [16] is a popular open-source library that simplifies WebRTC usage by allowing running an in-house signaling server, but does not implement any feature for decentralizing signaling. Gun-js [17], which aims to be a "decentralized database for developers", offers a peer discovery mechanism which can be compared to a federated rendezvous signaling server. However, it only supports native nodes. Support for a WebRTC-based DHT has been declared as interesting by the maintainers of Gun-js.

The two most prominent standardization efforts for DHTs are represented by LibP2P [3] and OpenDHT [2]. LibP2P [3] is a modular network stack that offers a Kademlia-like DHT as a "peer-routing" layer. OpenDHT [2] is a Kademlia-like DHT library with many bindings. However, these DHTs are built upon UDP thus they do not support browser technologies. Although LibP2P has early browser support, it is still in an embryonic stage as it does not support any discovery layer [18]. Currently, LibP2P browser discovery is only usable through some rendezvous servers that bridge native and browser clients. WebDHT designs variants of the same DHTs protocols to support browser and Web technologies.

WebTorrent [19] is a port of BitTorrent protocol to the Web using WebRTC as a P2P transport protocol. As WebTorrent targets Web environments, it is not compatible with traditional UDP-based protocols, but its WebRTC transport has been merged into libtorrent making it compatible with libtorrent-based native clients. Since WebTorrent can only use browser-compatible technology it cannot access the DHT network, and uses centralized WebSocket-compatible tracker servers.

Webrtc-explorer, also known as browserCloud.js [20], is a Chord-like DHT developed in 2015/2016. While the project is one of the first that aims to port a DHT to the browser it still needs a rendezvous server as it does not support WebRTC-based signaling. Detox [21] is the first project that claims to have a browser-compatible DHT. However, we could not reproduce their result as the project would not create browser-to-browser connections. P [22] is a pioneer project to leverage WebRTC as a signaling channel for WebRTC itself, however it does not include any decentralized routing logic. Nédelec et al [23] proposed specialized decentralized routing protocols for WebRTC. Although they propose theoretical analyses they do no propose implementations nor identity technical limitations of Web environments.

References

[1] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[2] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "OpenDHT: a public DHT service and its uses," in *Proc. 2005 Conf. Applications, technologies, architectures, and protocols for computer communications*.

[3] P. Labs, "Libp2p," https://libp2p.io/, accessed: 2022-12-05.

[4] R. Developers, "Rustcrypto: Nist p-256," https://github.com/RustCrypto/elliptic-curves/tree/master/p256, 2019, accessed: 2022-12-05.

[5] RustCrypto Developers, "RustCrypto: SHA-2," https://github.com/RustCrypto/hashes/tree/master/sha2, 2016, accessed: 2022-12-05.

[6] H. Boström, C. Jennings, and J.-I. Bruaroey, "WebRTC 1.0: Real-time communication between browsers," W3C, W3C Recommendation, Jan. 2021, https://www.w3.org/TR/2021/REC-webrtc-20210126/.

[7] P.-L. Ageneau, "libdatachannel," https://libdatachannel.org/, 2019, accessed: 2022-12-05.

[8] G. bug tracker, "Rtcpeerconnection objects are released too slowly," https://bugs.chromium.org/p/chromium/issues/detail?id=825576, accessed: 2022-12-05.

[9] ——, "Extremely slow ice gathering," https://bugs.chromium.org/p/chromium/issues/detail?id=1337554, accessed: 2022-12-05.

[10] J. Wärting, "Establish webrtc connection without a signaling server using nothing but web push," https://jimmy.warting.se/2021/02/16/p2p-signal-with-webpush.html, accessed: 2022-12-05.

[11] W3C, "Webrtc dht use-cases and proposals," https://github.com/w3c/webrtc-nv-use-cases/issues/15, accessed: 2022-12-05.

[12] L. Wang and J. Kangasharju, "Real-world sybil attacks in BitTorrent mainline DHT," in *2012 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2012, pp. 826–832.

[13] J. P. Timpanaro, T. Cholez, I. Chrisment, and O. Festor, "Bittorrent's mainline DHT security assessment," in *4th IFIP Int'l Conf. New Technologies, Mobility and Security*.

[14] T. Cholez, I. Chrisment, and O. Festor, "Evaluation of sybil attacks protection schemes in kad," in *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer, 2009.

[15] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based routing," in *Proc. 2007 IEEE Int'l Conf. parallel and distributed systems*.

[16] E. Z. Michelle Bu, "Peerjs," https://peerjs.com/, accessed: 2022-12-05.

[17] M. Nadal, "Gun," https://github.com/amark/gun, accessed: 2022-12-05.

[18] P. Labs, "Libp2p implementations," https://libp2p.io/implementations/, accessed: 2022-12-05.

[19] F. Aboukhadijeh, "Webtorrent," https://webtorrent.io, accessed: 2022-12-05.

[20] D. Dias, "browsercloud. js," *inesc-id*, 2015.

[21] N. Mokrynskyi, "Detox," https://github.com/Detox, 2018, accessed: 2022-12-05.

[22] O. Turgut, "P: peer-to-peer networking with browsers," https://github.com/unsetbit/p, accessed: 2022-12-05.

[23] B. Nédelec, J. Tanke, D. Frey, P. Molli, and A. Mostéfaoui, "An adaptive peer-sampling protocol for building networks of browsers," *World Wide Web*, vol. 21, no. 3, pp. 629–661, 2018.