The Backend of The Common Green: Kademlia-WebRTC Research Report

David Rincon-Cruz, Chengtian Xu

Columbia University, New York

Abstract

This research project was an exploratory attempt at creating a server-less "backend" for The Common Green, a peer-to-peer browser based social network. The benefits of such a service is the absence of "ownership" in rules and freedoms of speech. Without a single governing body controlling the flow of information in exchange for advertisements and selling private user information, there's no censorship obstructing truly free communication.

Knowing that this project requires a distributed database and runs on browses, the goal of the implementation of its backend is thus divided into two parts:

1. An implementation of Kademlia, a distributed hash table for decentralized peer-to-peer computer networks designed by Petar Maymounkov and David Mazires in 2002, and
2. WebRTC on top of Kademlia nodes as an interface for data transition.

We focused on implementing and merging them such that we would be able to create a distributed hash table on all active users with browser support.

Keywords: Peer-to-peer, Kademlia, WebRTC

1. Development

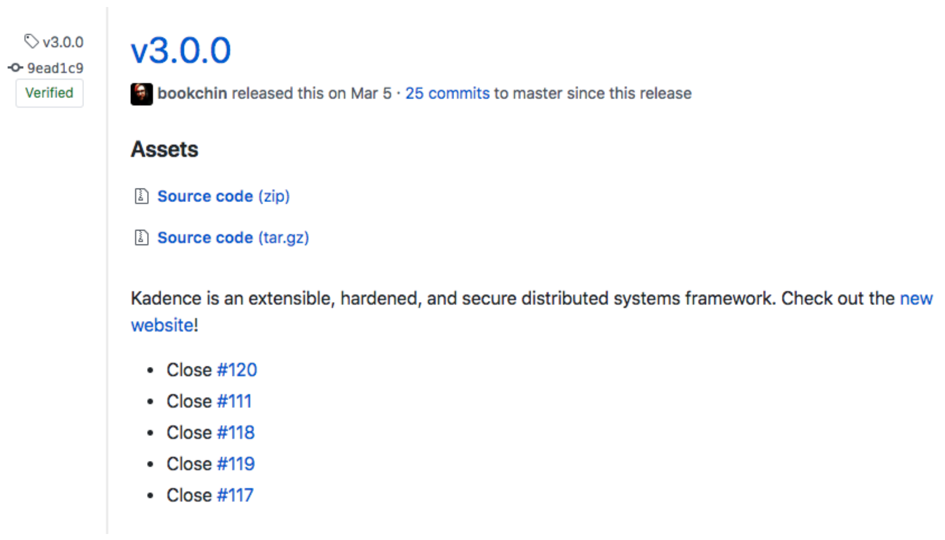
1.1. Prior to development

At the start of this project, we found a open source project **KadTools**, authored by Gordon Hall, under which there was *kad2.0*, an existing kademlia library implemented in JavaScript and runnable in browsers. Additionally, *kad-webrtc* was a plugin to *kad1.0* that attempted to utilize webrtc-connections as the transport layer of kademlia. However, by studying its source code we found that it did not maintain persistent WebRTC connections and placed heavy reliance on an external signal server. This negates the benefits of WebRTC connections entirely. If there's a need to constantly re-establish connections with a designated server, that server might as well just forward all messages.

1.2. Midway through development

Midway through development, the library had a new major release along with rebranding to kadence3.0.

Building our extension on top of *kadence3.0*, we developed *kadence-webrtc*tup as a transport plugin to kadence3.0. Additionally, as *kadence3.0* is not a library designed to run in browser due to native node dependencies, we've submitted pull requests to include browserify fields that make the project runnable in browser.


v3.0.0
9ead1c9
Verified

v3.0.0

bookchin released this on Mar 5 · 25 commits to master since this release

Assets

Source code (zip)

Source code (tar.gz)

Kadence is an extensible, hardened, and secure distributed systems framework. Check out the new website!

- Close #120
- Close #111
- Close #118
- Close #119
- Close #117

Figure 1: The 3.X.X release

Furtherso, as the end goal is to make a react application, this requires a compiled version of kadence3.0 and kadence-webrtc in ES5. This is a feature we are also working on.

1.3. Current Code

The layout of the classes is quite simple. A Kadence node utilizes WebRTCTransport as a duplexStream to read/write messages to. WebRTCTransport in turn owns Connection-Manager which manages mapping of nodeIds to WebRTCConnections, and decides where to write and push messages towards. For nodes with no existing connection, ConnectionManager utilizes classes implementing the CouplerAPI that output new WebRTCConnections for it to use. The current two are:

- SignalCoupler - Sends webrtc offers/answers through a dedicated websocket to a signal-server that redirects messages appropriately
- NetworkCoupler - Finds a communication path through the existing network of webrtc connections and by a system of forwarded messages, exchanges webrtc offers/answers through this channel until a direct connection is formed.

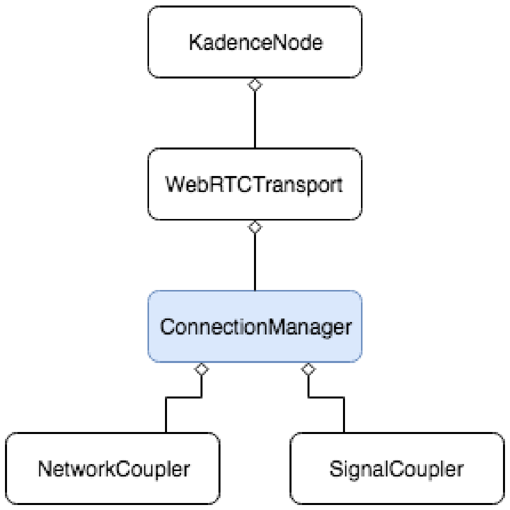


Figure 2: Diagram of Ownership of kadence-webrtc

2. In Progress

2.1. Coupler API

Two of the main classes in our package are called "couplers" as they create (or "couple") new webrtc connections between two nodes. The couplers are eventEmitters that emit new webrtc connections objects when offer/answer have been fully exchanged. Although the internals are different, a major step would be to generalize these two to share a more common api and utilize the same message format, allowing for the easy development of new couplers, as well as allowing nodes to attempt more than one in the case of failure. "TextCoupler" is a 3rd protocol that is in the works, allowing joining the network without ever contacting a signal server by relying on user-input to exchange the encrypted signal packets of webrtc offers/answers.

3. In Application

3.1. "Twitter" Demo

We were able to build a twitter-like message-board demo that utilizes kadence-webrtc by storing key-value pairs in a variety of manifests stored in the database. This emulates the way in which bittorrent stores an initial list of seeds, except the manifests are instead stored inside the database itself.

Every post contains three attributes: date, title, text. Stringifying the json object, we create a hash $H1$ and store it in the database. Then, for each word, w, in the title and text, we generate the hash corresponding to that word H_w. H_w maps to a object which is a map of dates to hash keys for posts containing w. Hence, the value of H_w is updated contain the property: post.date: $H1$.

Thus, querying for posts with a matching term, w, involves a single get request for H_w in the network.

3.2. Modifications To Be Made For Balanced Network Demands

Storing a very large value for a single key is discouraged since it demands unbalanced memory requirements across the network. The nodes near the location of large manifest are the nodes required to store copies, while other nodes send iterativeFinds for access to it. Hence, implementing a recursive, yet expandable system of manifest values would allow the system to scale up and balance memory requirements across different nodes.

key1 → {dates[1-100]: key2}
key2 → {dates[1-10]: key3}
key3 → {date1: post1, date2: post2, ...}

4. Scalability

The primary concern in this exploratory project, is the horizontal scalability of the mesh network created by users. At the kademlia level, every node ID and every key in the database is a 160-bit hash. Each node, has 160 buckets (one for each bit) with which it stores the closest k (default k=10) neighbors. Hence, at default capacity, each node can has 1600 neighbors. This is infeasible at the WebRTC transport level without heavy reliance on clever routing mechanisms or heavier reliance on signal servers.

Each browser maintains a different number of maximum WebRTC connections. For chrome, the limit is 256, however the true limit is dependent on throughput restrictions. Additionally, in the case of some of the packet-forwarding protocols, CPU also becomes a restriction with too many packets being sent through a node.

The kadence-webrtc package allows for specifying the maximum number of WebRTC connections a node can have, p, even if this number is less than 160*k. Whenever a node sends a packet to a neighbor with whom it does not share a WebRTC connection, it simply establishes one with the default coupler, stores it in a queue (maximum size p), and sends the packet.

However whenever p <160*k, this presents some problems in the long run regardless of the coupling method. Kadence nodes are set to ping their neighbors periodically to keep reliable neighbors and updated values stored. Pinging neighbors with which a webRTC connection does not already exists means having to re-establish WebRTC connections with the coupler. Using the signalCoupler, new connections are constantly made periodically, however it places a heavy reliance on the signal server for operation. Using the networkCoupler removes this dependency after joining the network, yet will result in network floods with ”find” and ”forward” signal packets carrying webrtc offers and answers.

The motivation behind persistent webrtc connections was to remove dependence on signal-servers, a feature not present in kad-webrtc. Using the signalCoupler re-instates this dependency. Using the networkCoupler removes this dependence but requires flooding the network unless stronger protocols are set in place.

Aside from reducing k, it will also be necessary to configure a way for webRTC connections to stay fixed (without a most-recent queue mechanism) and implement next-hop routing of messages. This will not only keep throughput high, but also reduce network floods of signal packets with webrtc offers/answers.

5. Reference & Documentation

For further references to source code/documentation, please refer to README from https://github.com/DRC9702/kadence-webrtc.