# WebTor

Nicholas Boucher, Gavin McDowell, and Raphaël Pestourie

Harvard University, REDACTED@college.harvard.edu, REDACTED@college.harvard.edu, REDACTED@g.harvard.edu

***Abstract* - Given the popularity and general success of traditional onion routing systems for web traffic such as TOR, we strove to implement an installation free, browser-based onion routing system. Dubbed WebTor, the system uses peer-to-peer communication channels to create relay circuits among peers who simultaneously act as both clients and relay nodes. In premise, WebTor allows anonymous internet browsing from any unmodified modern browser installation. The system is divided into three layers: the onion proxy layer, the onion routing layer, and the network later. While WebTor was not fully implemented, we made a number of interesting discoveries during the development process and provide a summary of those insights as well as general implementation details within this paper.**

*Index Terms* - WebTor, TOR, onion routing, peer-to-peer, WebRTC, JavaScript, Node

## Introduction

### *I. Motivation*

Onion routing systems were popularized through the creation of TOR[7,11,12] and have since been a popular method for anonymizing internet traffic. There are many legitimate motivations for wanting to anonymize internet traffic, including the protection of international freedom of the press, bypassing national firewalls, and avoiding government/ISP surveillance. Using the TOR network, however, can be quite complicated. The Tor Project has sought to make the process of using the TOR network easier with the creation of the Tor Bundle - a modified browser that is preconfigured to access the internet via the TOR network. However, even this method of accessing TOR can be too much of a burden for some non-technical users.

Our goal was to create a system that would make allow anonymizing web traffic via an installation-free, in-browser web application. This project, titled WebTor, hopes to open anonymized internet browsing to an entirely new set of non-technical users via its ease of use and platform reliability.

### *II. General Overview*

WebTor implements an in-browser onion routing system via a client-side javascript implementation that allows direct, peer-to-peer communication between users of the service. Each WebTor user serves a dual purpose within the WebTor network, playing the role of both a client and a relay node. That is, each user of the WebTor platform has the ability to request web traffic via the anonymous WebTor network. In addition, every user must serve (behind the scenes) as an intermediate relay node by relaying encrypted web traffic between other nodes. Optionally, a user can opt-in to being an exit node, which does then require the quick in-browser installation of a minimally-sized Chrome extension.

To simplify the implementation of this large project, we divided WebTor into three different component layers: an onion proxy layer, an onion routing layer, and a network later. Each of these layers is implemented in client-side JavaScript, with the exception of two additional components: an exit-node Chrome extension and a “bridge node” server, both of which are discussed later in the paper.

### *III. Layout of Paper*

This paper is divided into six main sections: (1) this introduction, (2) an explanation of the roles of each component layer of WebTor, (3) technical details of the Onion Proxy Layer, (4) technical details of the Onion Routing Layer, (5) technical details of the Network Layer, and (6) a conclusion.

## WebTor Components

WebTor is structured modularly into three different layers, each operating at a different level of abstraction. As shown in Diagram I, the topmost level is the Onion Proxy layer, the intermediate level is the Onion Routing Layer, and the lowest level is the Network Layer. This section will give a general overview of the roles of each layer.

### *I. Onion Proxy Layer*

The Onion Proxy Layer is the user-facing component of the WebTor application. It’s primary purposes are:

- To display a web interface which allows the user to indicate the desire to navigate to a web page via the WebTor anonymization network
- To capture, intercept, and redirect all HTTP fetch (e.g. GET and POST) commands preventing the default action
- To send all captured requests over the WebTor anonymization network via the Onion Routing Layer API
- To render and display all content received over the WebTor network within the browser to the user

The Onion Proxy layer is implemented using a combination of client-side javascript, html, and CSS. Our implementation made use of several experimental javascript web features which streamlined the interception of HTTP fetches, as described later in this paper.

*II. Onion Routing Layer*

The Onion Routing Layer is the middle-level component which implements the onion routing protocol. It provides the implementation of TOR-like routing system, agnostic of lower-level network details or higher-level display details. It's primary purposes are:

- To establish and maintain circuits within the WebTor network
- To handle circuit re-structuring when nodes drop out of the network
- To establish and maintain cryptographic implementations which secure communications between nodes

The Onion Routing layer is implemented using client-side javascript transpiled from ES6. It communicates using with the Onion Proxy and Network layers via their APIs, agnostic of their implementations.

*III. Network Layer*

The Network Layer is the low-level component which implements the peer-to-peer discovery and communication channels. It facilitates the instrument by which peer nodes/clients find and communicate with each other. It's primary purposes are:

- To create a mechanism by which peers in the network can find and discover each other
- To provide a channel for direct communication between specific peers
- To maintain a list of all active peers within the network

The Network layer is implemented using client-side javascript transpiled from ES6. It is called by the Onion Routing Layer via the Network layer's exposed API.

## Onion Proxy Layer

*I. Overview*

The Onion Proxy Layer provides a user-facing web interface while capturing, redirecting, and rendering all HTTP traffic through the WebTor network via the Onion Routing Layer. There were two particularly challenging technical aspects of the Onion Proxy Layer: first, finding a way to intercept all outgoing HTTP requests from within a given page in a browser window and second, finding a way to make outgoing HTTP requests from an exit node on the behalf of the requesting client.

As an added challenge, the goal in implementing this entire project was to not require the user to make any installations. We succeeded in that regard with the first task, and are thus able to use some experimental JavaScript features detailed below allow HTTP interception without any user action or knowledge. We were unsuccessful in this regard with the second task, as browsers go to extreme measures to ensure that user-level, client-side JavaScript cannot make Cross Origin HTTP requests without using the designated CORS paradigm. To circumvent these restrictions, we created a small Chrome extension that is required only when the WebTor user opts to act as an exit node.

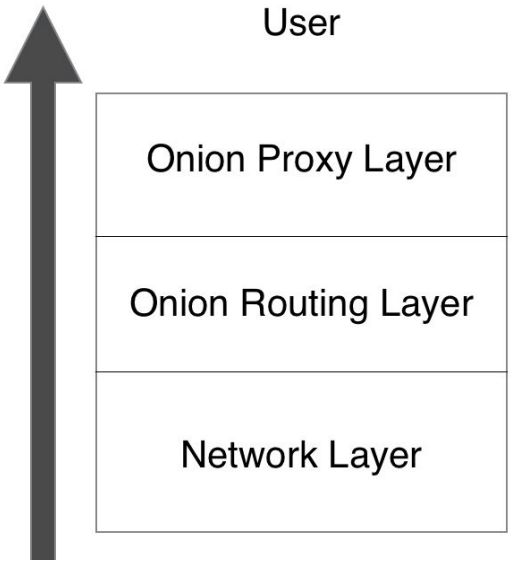


Diagram I: Layer Overview

*II. HTTP Interception*

The solution to HTTP Interception is one of the greatest sources of pride and accomplishment within this project, and can likely be used to implement arbitrary in-browser proxies outside of this project. We utilized a new, relatively unadvertised feature of browser JavaScript called ServiceWorkers[9]. ServiceWorkers are designed to help with caching data offline for large web apps that run within the browser. The ultimate goal is that they will replace the IndexedDB system that is present within many modern web browsers for caching purposes. ServiceWorkers work as follows, for any outgoing web fetch coming from a client that has a service worker *installed* for the relevant domain the respective ServiceWorker can analyze the request, determine if it already exists within the local cache in some form, and then optionally return the cached version to the user. *Installations* of ServiceWorkers happen silently and automatically, without user consent, if initialized properly within JavaScript.

We were able to "exploit" ServiceWorkers by using them to intercept all traffic coming from the WebTor client. We then simulate each request being cached locally, but instead of returning the non-existent cached version, we use Javascript to forward the request through the WebTor network via the Onion Routing Layer API. This effectively allows us to hijack any traffic coming from any page that we control, and silently redirect the traffic through a channel of our choosing.

To actually render HTTP fetch responses within the browser, we employ the use of iframes. This is convenient because iframes take care of html rendering for us in a manner that is relatively isolated from the container JavaScript/HTML code within the WebTor client application. The only technical detail that was that the src attribute for the iframe must appear to be on the same domain as the WebTor client application, or else the ServiceWorker will lose control over that web traffic. This is, however, easily circumventable by setting the iframe src attribute to something like "/fake-page.html?http://the-real-destination.com". The fake same-domain component of the URL is then easily parsed out by the Service Worker later.

*III. Exit-Node Web Requests*

Exit nodes presented another major challenge for us within this project. For obvious security purposes, all major browsers limit the kind of HTTP requests that arbitrary client-side JavaScript code can make to cross-domain sources. Specifically, it limits you to the CORS protocol. CORS will not allow us to fetch arbitrary web pages like the browser does, so we had to create a Chrome extension that allows us to make these "elevated-level" web requests. Specifically, we created a Content-Script [10] that gets automatically installed by the extension on any page within WebTor's domain. This content script exposes the ability to make arbitrary HTTP requests to the client-side JavaScript of Webtor.

Although this process requires the user to install a Chrome extension, we did not feel that this significantly changes the viability of the final product accomplishing the goal of the project. A user must only install the extension if they want to act as an exit node, which is entirely optional. In fact, we believe that it is unlikely the non-technical user who would be dissuayed from the product via installations would opt-in to become an exit node in the first place.

*IV. Implementation & Further Development*

The majority of the above mentioned code has already been implemented into our current code base using a combination of client-side and Chrome extension-based JavaScript. The code has yet to be tested though, because we were unable to finish implementing the channel by which the ServiceWorker communicates requests to the extension Content-Script. These scripts cannot communicate directly, and instead must communicate via PostMessages to the browser window. While this implementation is certainly non-trivial in difficulty, we are confident that it could be implemented successfully with more development time.

**ONION ROUTING LAYER**

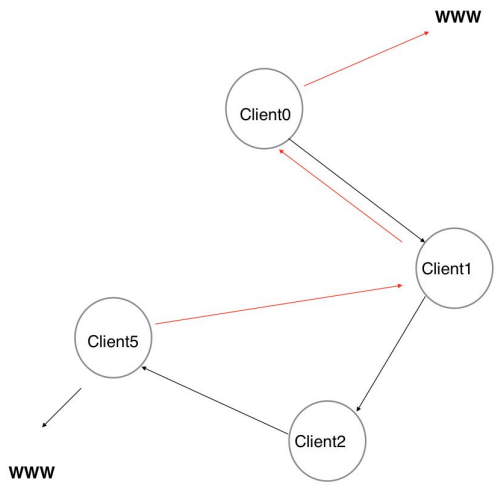


Diagram II: Onion Routing-level Circuit Overview

*I. Overview*

The Onion Router (OR) layer implements a Tor-like[7,11,12] overlay network on top of the network layer. It seeks to implement the primary semantic functionality of the original Tor spec according to the slightly modified needs of the Javascript environment in

which it lives. Intended as a demonstration object, many subtle technical details have been compromised on to make it more straightforward to discuss and analyze for the purposes of this discussion, but would not be reflected in a production artifact.

We can logically divide the router into several interdependent parts. The primary semantic separations are between the Circuit Abstraction, the Circuit Manager (enclosing OR scope), and the encryption layer. Diagram II is an illustration of the Onion-routing.

*II. Circuit*

The Circuit abstraction implements the primary Tor-based functionality under a javascript ES6 class abstraction.

*III. Circuit Manager*

The purpose of the Circuit Manager is to keep track of creating and destroying Circuits, choosing which peers to add, and extending the Circuits. It is primarily used as an interface giving the Onion Proxy Layer a higher level of abstraction to work with, primarily it exposes the "fetch" function to the Onion Proxy Layer, which handles the choosing of a circuit (or building on demand) to retrieve HTTP data for the OP to serve to the user. It also handles the asynchronous building and rebuilding of circuits.

I*V. Encryption*

We use two encryption protocols to ensure anonymity in the circuit. For the creation of the circuit we use a Diffie-Hellman exchange asymmetrically encrypted using RSA encryption algorithm.

The first node sends a create message with the first part of a Diffie-Hellman exchange that is RSA encrypted with the RSA public key of the second node in the circuit. The second node answers by sending a created message with the second part of the Diffie-Hellman exchange that is RSA encrypted with the RSA public key of the first node. Now the Diffie-Hellman exchange is complete and those two nodes will only communicate using symmetric AES encryption using the shared secret of the Diffie-Hellman exchange. The create message is semantically the same for the adding of each node in the circuit. The node that needs to send a create message receives an unencrypted extend message. The messages other than extend and create are relay message which we will cover later in this article. To provide a strong Diffie-Hellman implementation, we used a long prime number[13] and 2 as a generator.

Once the symmetric keys have been exchanged with every node in the circuit, the source node can communicate using only symmetric encryption. This encryption is layered, because each node adds a layer of encryption. This onion skin-like encryption is where the onion routing got its name from. The source node encrypts the messages starting with the symmetric key of the node for which it wants to send the message to, and then it encrypts with all the symmetric keys in a reverse order. When passing the message, each relay node decrypts the message before passing it along. The node the message is made for will decrypt with its symmetric key and and check a hash of the message (which is a property of the message). This node will see that the hash of the message corresponds to the property of the message that contains the hash of the encrypted message. That way the receiver node knows that the message is intended for it and will execute the command. If the property and the hash don't match, it means that the relay node has to decrypt with its symmetric key and send it along to the next node in the circuit. In the case of a request the exit node sees an unencrypted message, the property and the hash match, and the exit node does the request.

There exist different messages for the relay node: relay-extend, to send a create command to the next node in the circuit; a truncate command that will send a destroy message to the next node; and a destroy propagates.

Result of requests or relay-created message go towards the source. Each node encrypts the message with its symmetric key. When the source receives a message, it has to AES decrypt the message using the symmetric keys in the order of the circuit.

*V. Security*

We are not using TLS for the connection between the nodes within the circuit. Using only Diffie-Hellman and AES would leave the possibility of a Man-in-the-middle attack during the creation of the circuit. An attacker could pretend to be the next node and complete the Diffie-Hellman exchange. That is the reason why we add RSA asymmetric encryption. That way, the first half of the Diffie-Hellman exchange is encrypted, and the attacker cannot get the shared secret. (Note that we assume here that none of the nodes in the circuit are evil, which is a topic to examine in more depth for future iterations of this project).

This implementation exhibits perfect forward secrecy, which is that if an attacker knows public/private key pair of a node in the circuit and can sniff into the packets, the attacker still cannot know anything about the the packets unless they are the receiver of the packet, or an exit node, because of the onion-skin-like symmetric encryption. The exit node sees all the request in clear text, but does not know about the source. This is the reason why our Tor-like system is also anonymous.

Relay nodes don't know where they are in the circuit because all the create message are semantically the same.

*VI. Implementation*

The implementation of this section is far from complete. The demonstration functions of the *cryptography* module are implemented, and some of the basic Tor functions are implemented, but due to a lack of time for testing and a lack of capability of the network layer, these are largely unvalidated. In this same manner, the Diffie-Hellman exchange is implemented but also untested.

*VII. Further Work*

For the most part, further work on this part is a matter of finishing the current partial implementation, as there is nothing technical preventing this section from working successfully. Additionally, useful further work on this subsystem will also involve implementing a more complete feature set, closer to the actual Tor Project.

## Network Layer

*I. Overview*

The network layer comprises the lowest level interaction of the system, providing the mechanisms by which peers can establish connections and subsequently send messages to one another, along with a variety of other mechanisms. The network layer was designed to use primarily the WebRTC[1] protocol, which requires the use of signaling channels which will be discussed in greater depth in this section.

*II. WebRTC*

The system was designed around the use of WebRTC as its primary underlying communication layer, relying on a peer-to-peer mesh for signaling, as well as direct connections for circuits. WebRTC is a recently-developed technology which allows for the establishment of a direct channel communication between two browsers behind unrelated NATs. The technologies it uses to do so are STUN[2], ICE[3], and TURN[4]. STUN and ICE only require the passing of signaling messages between the two peers, and are thus usable for our purposes. TURN instead relies on an external server to which both peers establish independent connections. The TURN server will then relay messages from one to the other. Its lowest level network layer is the Secure Real-time Transfer Protocol[5] (SRTP), which provides good low-level encryption and authentication options which are utilized by WebRTC for secure low-level communication. Our implementation does not trust this, however--as discussed earlier we still use the onion routing technique for nested encrypted channels to deal with malicious relays not just MITM attacks.

*III. Signaling*

The signaling mechanism is where most of the effort went in the development of the network sublayer, as its implementation is critical to the robustness and usability of the network. WebRTC does not define a particular signaling mechanism, and instead relies on the existence of arbitrary alternate signaling mechanisms to pass both Session Description Protocol (SDP) messages and Internet Connectivity Establishment (ICE) candidates between the two peers. In practice this is often facilitated primarily through a centralized signaling server of some kind. In the design of WebTor we focused primarily on being able to rely as little as possible on any centralized signaling mechanisms. For this reason, the primary signaling mechanism is via a routing algorithm over a WebRTC peer mesh. Each relay node instance (browser tab on the website) will generate its own unique ID. This identification number serves as the unique identifier for a relay node in the network.

*IV. Signal Routing*

The signal routing mechanism is designed to treat the aforementioned unique id's for each relay node as locations in some space, and thus uses a distance-function routing algorithm. In particular, when a relay node receives a signaling message which is determined to not be intended for it, it will compare its intended recipient with the list of peers to which it is maintaining open signaling channels, and forward it along the channel corresponding to the peer with the closest *id* to the intended recipient. This is similar to a routing mechanism in a content-addressed distributed file system, and relies on a certain set of network properties to work well.

*V. Network Properties*

In order for the signal routing to work, there are a number of network properties that need to be maintained. We do not have a rigorous proof that even under these network conditions the system will be guaranteed to succeed, but we predict that if these network properties can be well-maintained, then it will successfully deliver signals to their intended recipients in almost all cases.

First, it is absolutely necessary that the network not be partitioned. We don't have a good way of maintaining this property, so instead we rely on detection of disconnected peers and simply treat the partition of a user as the entire network for them. Thus we can assume this property is maintained.

The way we attempt to maintain the property that nodes in the network be findable according to their *id*s, assuming a fully connected network, primarily revolves around demanding that each node maintain an open connection to the *k* known peers with *id*s closest to the node's own *id*. The

value of *k* would have to be tuned experimentally. Additionally, it is likely that after experiment we would hope to optimize the routing in the average case, and to do so we would add to this open connections to *j* random other peers evenly distributed across a broad range of *id* values.

*VI. Peer Discovery & Lost Peer Detection*

A nontrivial aspect to this network is the issue of peer discovery in the absence of an abstraction akin to the Directory Server of the Tor network. The key to our design is that we completely eschew the concept of consensus, opting to rely on our signaling network topology to provide Pretty Good Consensus which propagates outward from a node's entry point, moving as quickly as possible to the region of *id* space closest to a peer's own *id* in the interests of maintaining our network properties. As such, peer discovery is done strictly through peer-to-peer broadcasting over the signaling mesh. When a peer connects to a previously unknown peer, it will immediately broadcast a SIG_PEERLIST message to all of its connected peers containing at least the newest added peer. Each entry in a peerlist will contain at least an RSA public key for that peer (from which the peer's id can be derived).

In order for the network to be robust responding to churn, we need a method of detecting and quickly broadcasting that a peer has been lost. When a peer's connection to another peer closes unexpectedly, it will generate and broadcast a SIG_BAD_PEER message to all its remaining connections. Each relay node will maintain a list of bad peers that it will refuse to route to until it receives a SIG_NEW_PEER message for that peer. This means that as this message spreads, it will mean that routing to this peer will fail more and more quickly making the network become more and more responsive to missing peers. If a node receives a SIG_BAD_PEER message for a peer it has an open connection to, it will broadcast a SIG_NEW_PEER message for that peer to tell the network that that peer isn't lost. This message will propagate out and re-allow routing to that peer over the network.

Note that each relay must also keep a record of messages it has broadcast and never rebroadcast a message twice. This prevents infinite rebroadcasting which would start overwhelming the signaling mesh.

*VII. Bootstrapping*

While the routing mechanism discussed above is largely sufficient for maintaining network properties and building circuits once connected to at least one other node in the mesh, it still doesn't establish a mechanism by which a new peer may join the network. For this we require alternate bootstrapping mechanisms, and provide two, serving different goals.

The first goal is general-case convenience. If we make the nontrivial assumption that the webserver providing the proxy/relay application is non-malicious, and that we as the developers could notice and take things down if they were compromised, then we can provide a signaling server which runs in node and allows for the most convenient method of establishing a connection to another node in the network. As originally intended, what we have termed a "bridge node" also serves as a generic relay node. It maintains the same network properties as discussed above as any other relay node, and implements the same functionality. This allows for an easy entry point for joining the network.

The second goal is worst-case reliability. The intention is to provide a fallback mechanism in case the bridges are down, compromised, or untrusted, providing a completely out-of-band method of attaching to the mesh. Unfortunately, this requires two active partners--one of whom must already be in the network--who have access to some out-of-band communication method by which they can share base32 strings with one another. This is referred to as manual signaling, and allows two peers who wish to connect to establish a connection without any reliance on external servers whatsoever. They both startup the manual signaling system and send base32-encoded binary signals to each other as they are shown on screen by the WebTor application and copy-paste them into their browsers until both have enough data that an RTCPeerConnection can be established, which requires a minimum of three signals.

*VIII. Security*

STUN and ICE require sending packets including raw IP addresses to each other. This is not necessarily a bad thing, especially as in Tor this information for a relay was necessarily public knowledge. This thus provides the same privacy properties for relays as the original Tor network. The primary advantage of this is how lightweight the relay is, requiring no install, as discussed previously. We also maintain the secrecy properties of the Tor network with regard to the circuit, as no node in a circuit need be aware of the circuit topology, nor the identities of any other nodes in the circuit except those adjacent to it, to which it must naturally have direct connections.

It seemed at first that TURN could provide an advantageous way to hide the IPs from one another, but this turns out to be a net loss of security, as the reliance on the TURN relays would create an obvious attack vector which would basically provide universal knowledge of the mesh network through compromised TURN relays, so we eliminated TURN as a possibility.

The security of the signaling network is also a serious concern. It would seem that many nodes in the network would be able to easily correlate data and keep track of who is connection to whom, but we believe it to be easy to

obfuscate this sufficiently to satisfy the concern. In particular, one major concern may be mitigated by the use of asymmetric encryption for signaling payloads. Thus, while the destination id for any signaling message needs to be in plaintext throughout for the routing mechanism to function, the source of the signaling message can be encrypted using the public key of the intended recipient as part of the payload. This, combined with the expected fairly high rate of signaling messages related only to the churn of the signaling network, means that correlating packages to determine circuit it topology would be an incredibly difficult problem to solve, though we have not proven its impossibility.

*IX. Implementation*

Some of this was implemented. In particular, much of the underlying signal layer was implemented. It uses a callback-based architecture which makes its interface simple, but has the unfortunate property of making it difficult to analyze and debug. This effect is common in JavaScript and is generally referred to as "callback hell." This is one of many problems that has plagued the development of this project, in spite of our attempts to alleviate the pains of JavaScript using various ES6 encapsulation features. This is certainly one place where significant improvements could be made. However, the manual signaling mechanisms are working as expected, and RTCPeerConnections can be established successfully through out-of-band signaling. It is expected that the direct peer signaling mechanisms could be made to work in short order.

The main sticking point of this section was the implementation of the "bridge nodes." Intended to run as full relay nodes, they require a Node implementation of WebRTC which was assumed to exist, and certainly can exist in theory. In practice, expectations are often disappointed, and we were unable to make the most developed Node package attempting this to work in our environments in the time that we had. Thus, in its current state, the implementation of "bridge nodes" is instead as a glorified signaling server which allows peers to connect looking to establish a connection to any other peer. In response to a SIG_REQ_PEERLIST message, it will send a list of other peers that it is connected to. The client may then pick a peer from this list and then try to open a channel using the bridge node as a signaling channel. The bridge node will forward any signals that it can between nodes that it is connected to. The bridge in its current state is forwarding messages correctly, but the bridge signaling is currently suffering from an ambiguity in the use of the signal *id* field which would be straightforward and relatively quick to correct, but which was abandoned in favor of the completion of this technical discussion.

The other primary function that there was not time to develop was the actual routing algorithm. The mesh routing system as described above is thus unvalidated. This is discussed further below.

Additionally, as most distributed systems and most Javascript are wont to do, our implementation suffered greatly from a difficulty in debugging and would be well-served by a few days of development strictly on a stronger debugging and logging system and a more careful distributed design.

*X. Further Development*

There are obviously many ways that the current state of the project could be improved, both in implementation and design.

The "bridge node" architecture is also currently inadequate, as it is largely serving as a glorified signaling server build on the remnants of something that was designed to serve as a full relay node. As such, a complete redesign of this general-case bootstrapping mechanism is worth considering to reconcile its design with its implementation. Alternately, reimplementing it when there exists a satisfactory WebRTC implementation for Node is a worthwhile consideration.

The signaling network bears closer scrutiny and experimentation. It may turn out that under experimentation, this arbitrarily constructed routing algorithm proves to be wholly inadequate in one or many ways. If this turns out to be the case, we would probably turn to a more tried and tested routing algorithm and attempt to adapt it to work for our peer mesh network. Probably the first step of this would be to attempt to implement a tried and tested routing algorithm designed for low-churn networks over our peer mesh, such as IEEE 802.1aq[6] but it is unclear whether this would be sufficient in a network with a nontrivial expected churn. There would in all likelihood be a large amount of effort spent in making this work well. It cannot be emphasized enough that this is a difficult problem that does not currently have even a particularly Good solution in the literature, as most routing algorithms with significant academic work behind them are those designed for networks with low churn, and thus focus on fast convergence to shortest paths. Short of probably years of academic study working on this problem, we have opted for a more heuristic solution, attempting optimizing parameters for a good average case, absent any theoretical guarantees. Even this, however, would take a significantly longer time than was available to us.

## Conclusion

We believe that it is possible to create a working version of an entirely in-browser, installation-free onion routing system which communicates via peer-to-peer

protocols and in which clients also act as relay nodes. While we did not have enough development time to provide a full, working version of this product, we believe that this implementation of WebTor could be completed successfully with more time.

Furthermore, we discovered multiple things during the partial implementation of WebTor, including a signaling algorithm that we believe will allow peer-discovery over a distributed mesh network and an in-browser web proxy solution that cleanly and silently intercepts and redirects all web traffic from a given domain and window. Therefore, while there is still significantly more work required to accomplish the initial goals of WebTor, we as the developers are happy with the amount of information that has been made known simply by the partial implementation of this system.


### Acknowledgments

The authors of this paper would like to acknowledge the contributions of Professor James Mickens, Michael Crouse, and Jerry Ma via the materials taught in Computer Science 263 at Harvard University.


### In Text References


[1] World Wide Web Consortium (W3C), "WebRTC 1.0: Real-Time Communication Between Browsers, "https://www.w3.org/TR/webrtc/, 24 Nov 2016

[2] Rosenberg, J., et al., "Session Traversal Utilities for NAT (STUN)", RFC 5389, Oct 2008

[3] Rosenberg, J., et al., "Internet Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245, Apr 2010

[4] Mahy, R., et al., "Traversal Using Relays around NAT (TURN): Relay Extensions to STUN," RFC 5766, Apr 2010

[5] Baugher, M., "The Secure Real-time Transport Protocol (SRTP)," RFC 3711, Mar 2004

[6] "802.1aq - Shortest Path Bridging," IEEE 802.1aq, 29 Mar 2012

[7] The Tor Project, Inc., "Tor Project: Anonymity Online", https://www.torproject.org/

[8] Mozilla Developer Network, "Service Worker API", https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API, 23 Aug 2016

[9] Matt Gaunt, "Service Workers: an Introduction", https://developers.google.com/web/fundamentals/getting-started/primers/service-workers, 30 Nov 2016

[10] Google, "Content Scripts.", https://developer.chrome.com/extensions/content_scripts, 02 August 2013

[11] Dingledine, R. et al., 'Tor: the second-generation onion router', Proceedings of the 13th, conference on USENIX Security Symposium, p.21-21, August 2004

[12] Dingledine R., "Tor Protocol Specification", https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt

[13] Kivinen, T et al., "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)",SSH Communications Security, May 2003


### Author Information


**Nicholas Boucher,** Student, Harvard College, Harvard University.

**Gavin McDowell,** Student, Harvard College, Harvard University.

**Raphaël Pestourie,** Student, Graduate School of Arts and Sciences, Harvard University.