Peer-to-peer Browser Middleware for Communication and Consensus in an Unreliable Infrastructure

Joel Martin
Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76010
Email: joel@martintribe.org

David Levine
Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76010
Email: levine@cse.uta.edu

***Abstract*—We describe a browser-based peer-to-peer computing system/middleware that enables distributed consensus that is tolerant of degraded and/or unreliable centralized infrastructure. The system is composed of a JavaScript implementation of the Raft distributed consensus protocol (Raft.js) together with the WebRTC browser-to-browser communication protocol. The Raft over WebRTC middleware is used to build a demonstration chat application that guarantees a shared, consistent and linearized view of the chat history without relying on a single centralized server for communication. The correctness, robustness, performance, and scalability of the system is verified with a series of automated tests implemented using the SlimerJS browser automation tool.**

1. Use Case/Scenario

Consider a scenario where emergency responders and volunteers are responding to a city-scale disaster in which utilities such as electrical power and communication systems are highly degraded or completely unavailable for an extended period. Emergency management officials have divided the city into zones each of which is assigned several dozen volunteers who systematically search the neighborhoods for people that are injured or in need and also for damaged infrastructure that needs to be repaired.

Responders and volunteers have access to a distributed disaster response datastore that is accessed and updated via a browser based application. Volunteers use their own mobile device and only require a browser with WebRTC capability to run the application. The application communicates over the cellular network (existing or provided by mobile emergency response cell stations) or mesh network (although this may also require additional installed software on mobile devices in order to support it). Once loaded and running, the application does not require any centralized servers or database system. The application uses the Raft distributed consensus protocol to maintain a distributed but consistent view of the datastore among all participating client nodes. Also, the distributed cluster is dynamic and allows nodes to join and leave over time without losing state or synchronization of the datastore.

The above capabilities are demonstrated using a distributed chat application implemented with the Raft over WebRTC middleware. This chat application uses direct browser-to-browser communication over WebRTC and maintains a consistent view of the chat history (exact same messages and ordering is seen by all nodes) and is fault tolerant (almost half the cluster nodes can fail simultaneously and the application will remain available without dataloss for the remaining clients).

2. Background

2.1. Raft

Raft is a new distributed consensus algorithm designed to be understandable and practical without sacrificing safety and correctness. The Raft algorithm implemented and described in this paper is based on Diego Ongaro's dissertation "Consensus: Bridging Theory and Practice" [1].

The Raft algorithm breaks the problem of consensus into three core sub-problems: leader election, log replication, and safety [1, Section 3]. A full Raft system must also address: cluster membership changes [1, Section 4], log compaction [1, Section 5], and client interaction [1, Section 6]. The Raft protocol is based on the concept of a distributed transaction log. Entries in the log represent a sequence of commands that will be applied to an internal state machine. If all members of the Raft cluster have the same log entries, then their state machine on each node will have exactly the same state. Each node of a Raft cluster can be in one of three states: follower, candidate, or leader. The responsibility of the leader is to accept new transaction log entries and then replicate these entries in the same order to all other members of the cluster. The leader is the only member of the cluster that may make changes to the transaction log. In order to maintain leadership, the leader sends heartbeat RPCs to all the followers in the cluster. Followers that do not receive a heartbeat RPC within a certain time period become candidates and attempt to be elected by the other nodes as the new leader of the cluster [1, p. 12].

Raft leverages the replicated transaction log to accomplish live cluster membership changes. Membership changes are accomplished by adding or removing one Raft node at a time using special add/remove log entries. In order to keep the replicated transaction log from growing indefinitely, Raft nodes should periodically compact their logs. The simplest and most generic solution is often to provide some way of snapshotting the entire state machine (including the term and log index it represents) to disk at which point the all previously applied log entries can be discarded. Clients of the Raft cluster are able to interact with the distributed state machine by sending RPCs to the leader in order to add command entries to the transaction log. Clients first find the address of any node in the cluster and are then redirected to communicate directly with the leader. The Raft protocol also provides strict linearizable semantics for client commands/requests (reads and writes) for applications that require it.

2.2. WebRTC

WebRTC is a collection of browser APIs [2] and network protocols [3] that are currently being standardized by the W3C and IETF respectively. Collectively these APIs and protocols enable real-time peer-to-peer video, audio and data communication between browsers.

One of the challenges with browser-to-browser communication (and direct peer-to-peer communication in general) is that the browser is usually running behind a firewall in a private internet subnet. To establish a direct communication channel between browsers, WebRTC uses a mechanism called Interactive Connectivity Establishment (ICE) that is similar to the Session Initiation Protocol (SIP) commonly used for Voice-over-IP (VoIP) communication [4]. This process requires out-of-band exchange of information usually involving a third party server known as a signaling server. The signaling server may optionally provide other services such as higher level session management, HTTP serving of the web application, address book service, client presence/availability, etc.

2.2.1. WebRTC APIs. The WebRTC browser APIs enable JavaScript web applications to establish browser-to-browser network connectivity in order to perform real-time video, audio and data communication. These APIs are currently in the process of being standardized by the *Web Real-Time Communications Working Group* of the World Wide Web Consortium (W3C) [5].

There are three primary WebRTC interfaces: *RTCPeerConnection* [6] for establishing direct connections between browsers, *RTCDataChannel* [7] for sending and receiving generic data between browsers, and *MediaStream (aka getUserMedia)* for managing audio and video streams (not relevant to Raft over WebRTC). The RTCDataChannel API supports both multiple modes for reliability and ordering which allows average latency/jitter to be decreased at the cost of some messages being dropped or arriving out of order. The Raft protocol is specifically designed to support dropped or out-of-order messages so the transport options

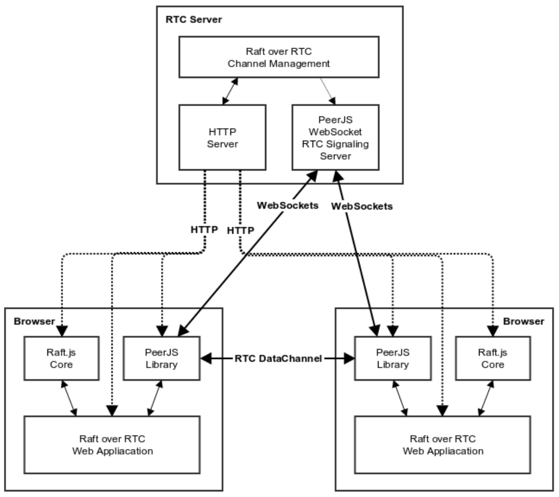

Figure 1. Raft over RTC Architecture

available with RTCDataChannel can be used to gain extra efficiency.

2.2.2. WebRTC Protocols. There is a large suite of protocols that are required for a working WebRTC system. The Internet Engineering Task Force (IETF) organization is responsible for specifying and/or standardizing (where necessary) the protocols that are part of WebRTC [3] [8] [9]. In many cases an existing protocol is applicable and is referenced or extended as part of the WebRTC suite. The protocols that make up the WebRTC suite are categorized into five general functionality groups: data transport, data framing and security, connection management, data formats, presentation and control, and local system support functions. The last three groups focus on interactive audio and video applications and are not relevant to Raft over WebRTC.

3. Design and Implementation

3.1. Raft.js Core Library

Raft.js is an implementation of the Raft algorithm in JavaScript that was created by the author of this paper [10]. Raft.js is designed to run in either a browser environment or within Node.js (server-side JavaScript). Raft.js implements the Raft algorithm as described in [1, Consensus: Diego]. The core Raft algorithm is defined in the `RaftServerBase` abstract class (in `base.js`) which omits the implementation of methods with side-effects such as RPC communication, durable storage, scheduling/timeouts, or state machine command/management. The full instantiation of the class (in `rtc.js` and `chat.js`) adds state machine commands and the ability to send RPCs over the WebRTC Data Channel.

3.2. PeerJS

PeerJS is a project that provides a simplified abstraction for using WebRTC. The first component of PeerJS is a node.js server library that implements a WebRTC signaling server. The second component is a JavaScript library that interacts with that signaling server and also provides a simpler interface for using the WebRTC APIs.

3.3. Signaling Server

The file ‘rtc_server.js’ implements the signaling server for the Raft over WebRTC application. The server extends the PeerJS ‘ExpressPeerServer’ object in order to provide WebRTC signaling. In addition to the WebRTC signaling function the server also serves static web resources (HTML, CSS, and JavaScript) and WebRTC channel management endpoints.

3.4. Client

The Raft over WebRTC chat application is a single web page (`chat.html`). When a new Raft cluster is created, the browser connects to the signaling server’s new channel endpoint (Figure 2 [1]). This endpoint causes the server to create a new channel for PeerJS signaling (Figure 2 [2]). Once this is complete the server responds to the browser with a redirect message that contains the new channel identifier in the query parameter and “firstServer in the fragment identifier to indicate that this is the very first Raft cluster node (Figure 2 [3]).

Subsequent nodes are added to the same Raft cluster by starting a new browser context (separate browser, browser window, or browser tab) and loading the same URL without the “firstServer” fragment identifier (Figure 2 [9]). There is a convenience link provided in the application page that opens a new browser window with a new cluster node.

When each page first loads, a PeerJS signaling connection is established with the server (Figure 2 [6] and [11]). The web application also begins a periodic async polling function (`addRemoveServersAsync`) that compares the current active list of PeerJS clients to the Raft cluster membership list. If there are any differences between the two lists, then any new clients are added to the Raft cluster and missing clients are removed from the Raft cluster one at a time using the `addServer` and `removeServer` RPC calls. Servers additions are prioritized before removals in order to maximize availability [1, Section 4.4].

When an instance of the web application starts that is not a “firstServer”, the normal Raft timers are not enabled until the server receives its first RPC. The prevents the server from trying to becoming a leader before it is incorporated into the cluster by an appendEntries RPC from the true cluster leader.

3.5. Chat Application

The chat application extends the basic Raft.js over WebRTC client. The chat application provides a large text area containing the chat history and a text entry field below it. When the user clicks on the *Send* button (or presses enter), the current node ID is prepended to the beginning of the text line and it is added to a pending send queue and the text entry field is blanked. The chat application also extends the default set of state machine operations to add a *seqAppend* operation. This is an atomic operation that is implemented as follows:

1) The *seqAppend* operation operates on a specially structured key/value pair in the state machine map. The value of the pair is another associative map that contains a counter and list structure where the actual chat lines are accumulated. *seqAppend* operations are sent with both a count and then new value to append.
2) When a *seqAppend* operation is applied to the state machine and the count value in the command matches the counter in the state machine’s value, then the new line in the request is added to the value list and the counter is incremented. If the count value does not match the counter then the the operation is not applied. In either case, a list is returned in the `clientRequestResponse` that contains two values: a Boolean indicating whether the *seqAppend* operation succeeded and a number indicating the current value of the counter.
3) The first time the *seqAppend* operation is used on a non-existent key, the structure described above is created with the counter set to 0 and the list structure is created with the single value that was passed in the *seqAppend* operation.
4) Clients that call the *seqAppend* operation should

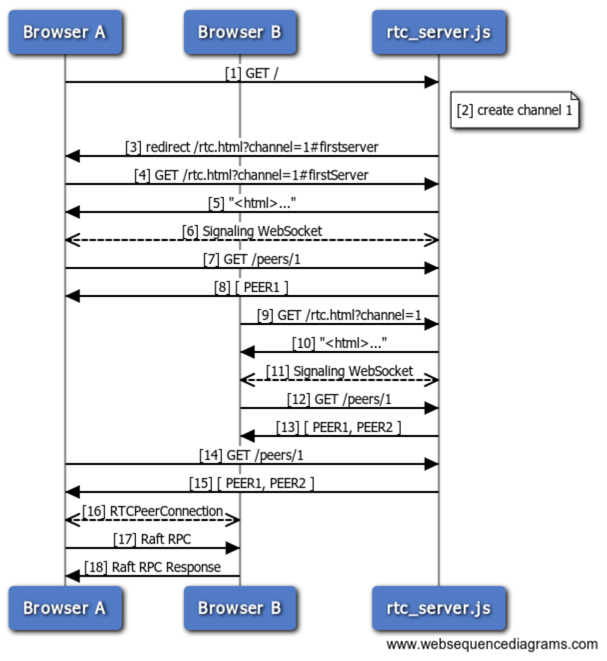

Figure 2. Raft over RTC Sequence Diagram

check the status value in the result list. If the status is false, then the client should retry the operation. Regardless of success of failure, then client should store the current value of the counter for the next *seqAppend* call or retry.

4. Experimental Results

4.1. Test Setup

Automated testing of the Raft over WebRTC system was performed by leveraging the SlimerJS scriptable browser environment. SlimerJS enables web applications to be scripted by loading pages in a special Mozilla rendering environment called XULRunner. The XULRunner instance is directed to an X virtual frame buffer (xvfb) which provides effectively headless testing.

To simplify the execution of the tests and to make the testing infrastructure more portable, a docker container environment encapsulates and executes the SlimerJS environment. The container is built using *test/Dockerfile* which is in turn derived from the "fentas/slimerjs" docker image [11]. This custom Dockerfile was created because the upstream version of the Dockerfile used an older version of XULRunner that did not support more than 7 WebRTC clients communicating together simultaneously. This issue was discovered during initial testing.

Writing tests using SlimerJS involves creating JavaScript test programs which start a server instance and then launch a SlimerJS docker container for each node of the cluster. Each container starts a XULRunner browser instance of the Raft over WebRTC chat application. Once all the chat application nodes have connected to the test server/program, then the actual test is started across all the application nodes.

Three different tests programs/servers were created to test the basic Raft over WebRTC application. Each of the tests takes a parameter indicating the number of nodes/pages to launch. The test source files are:

1) *test/common.js*: a module of common functions used by the other tests scripts.
2) *test/test_up.js*: launch and start an initial leader node/page, then start the rest of the nodes (if any) so that the leader will configure them into the cluster. Then test then waits for every node to be recorded (applied) into the log of every other node of the cluster. The test reports the number of milliseconds for the cluster to reach this state.
3) *test/test_kill_nodes.js*: similar to *test/test_up.js* but once all the nodes are configured into the cluster, this script then kills the leader plus a configurable number of nodes. By default the script kills just under half the nodes of the cluster. The test then waits for a new leader to be elected and remove the dead nodes from the cluster. The test reports both the number of milliseconds for the cluster to come up and the number of milliseconds for the cluster to fully recover from the failure and evict the failed nodes.
4) *test/test_chat_propagate.js*: similar to *test/test_up.js* but once the cluster is up, the test connects to one of the non-leader nodes and and sends a chat text line. The test then waits for the chat message to propagate to every other node's state machine. The test reports both the number of milliseconds for the cluster to come up and for the messages to propagate to every node.

The tests were all run on a system with 8 Intel Core i7 processors running at 3.20GHz and 12 GB of main memory. The Docker image used to run the test was built using Ubuntu 12.04.5 LTS with *SlimerJS* version 0.9.6, and *XULRunner* version 38.0.5.

4.2. Outcome

Figure 3 and Figure 4 show the result of running the *test/wait_chat_propagate.js* test to determine the amount of time that a stable cluster takes to propagate, commit and apply 1 and 100 chat message commands respectively to the state machine on every cluster node using the atomic *seqAppend* command. The commands are serialized which means that the previous command must be committed before the current command can be committed. The test is complete once the last chat message command has been both committed and applied to every node of the cluster. Each test execution is shown as a blue "x" and the black line represents the average for each cluster size.

In Figure 3 the time to propagate a single message to a single node of the cluster is approximately 0.1 seconds. However, this really just represents the tests scanning/checking granularity. The time to finish propagating and committing a single message shows a slight linear slope upwards from about 0.3 seconds for a three node cluster to about 0.5 seconds for a 41 node cluster. The time to propagate 100 messages (Figure 4) has a much higher slope and also appears to be non-linear.

Figure 5 shows the results of running the *test/test_up.js* test to determine the amount of time that a cluster takes to start. The cluster is fully started once a leader has been elected and the leader has successfully added all the other nodes to the running configuration. Each election timer value is represented as a different color and the line represents the average value for each cluster size. The cluster startup times show a fairly linear increase as the cluster size increases. The reason that the slope varies with election timer is due to the fact that the election timer also determines the leader heartbeat value (currently set to one fifth of the election timer value). The leader heartbeat determines the frequency with which *appendEntries* messages are sent from the leader to followers and is also used to propagate log entries. This means that the rate at which log entries (and thus cluster configuration updates) can be propagated is proportional to the election timer value.

Figures 6 and 7 show the results of running the *test/test_kill_nodes.js* in both modes (kill only leader and

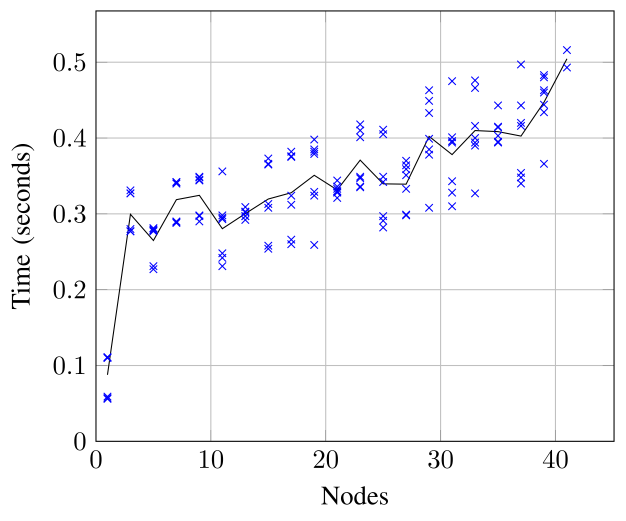

Figure 3. Propagate, Commit, and Apply 1 Command

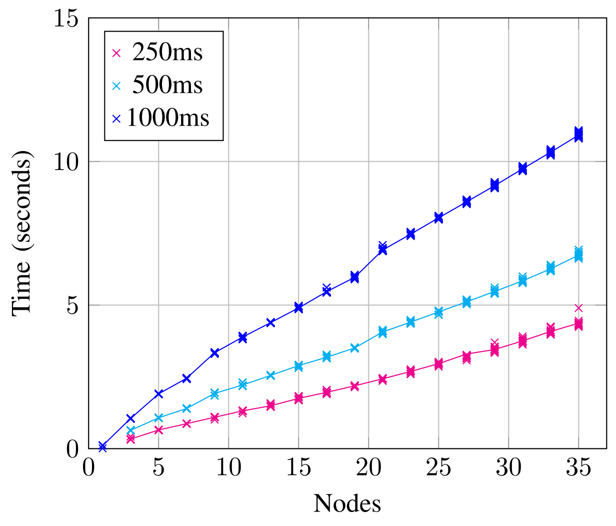

Figure 5. Initial Cluster Activation

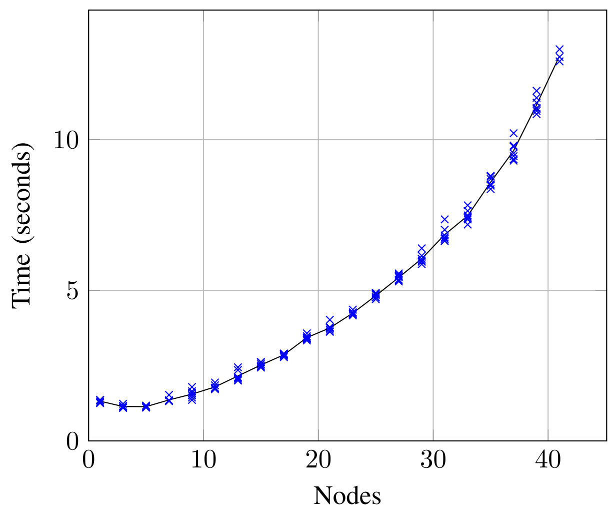

Figure 4. Propagate, Commit, and Apply 100 Commands

kill just under half the nodes). Each election timer value is represented as a different color and the solid line represents the average recovery time for each cluster size. When only the leader is killed, the recovery times are constant flat regardless of cluster size. This is because after a new leader is elected, there is only a single configuration entry to propagate to the rest of the cluster (to remove the old failed leader). One interesting thing to note is that with a 1000ms timer, the 3 node cluster is actually the most inefficient size because the average time until a leader election starts is inversely proportional to cluster size so the delay until an election happens is a significant contributor to the three node cluster with 1000ms timer. With a larger cluster or shorter election timer, the contribution of election delay is not noticeable. The recovery from almost half the nodes failing is not flat because the amount of configuration information that needs to be propagated is proportional to the size of the cluster.

5. Future Work

One of the useful outcomes of this project was that it brought to light many areas where further study of Raft, WebRTC, and Raft over WebRTC should be pursued. Here are some interesting areas for further exploration:

1) Explore models for dynamically adjusting Raft timeout values to account for changing network and browser conditions.
2) Explore alternate WebRTC transport modes for relaxing ordering constraints and delivery guarantees. Since the Raft protocol is tolerant of out-of-order and dropped messages, it is possible that Raft over WebRTC could be more efficient with one or both of these constraints relaxed.
3) Test the survivability of the cluster when the signaling server becomes completely unresponsive or very slow.
4) Test larger scale deployments spanning larger and more diverse network domains. Characterize the factors that affect and limit scalability.
5) Explore models for timing out and automatically removing nodes when they become unresponsive without needing to be notified by the signaling server that the nodes have disconnected.
6) Consider extensions to the Raft protocol that enable slow leaders to be proactively detected so that a more capable node (network, CPU, etc) can be elected. This would help address that background page/tag throttling of modern browsers.
7) Explore models that allow for a combination of strict shared state and high performance bulk data such as media or object blobs.

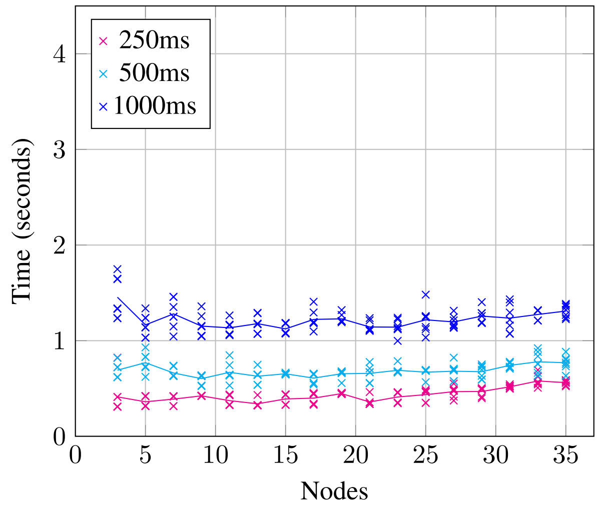

Figure 6. Recovery from Leader Failure

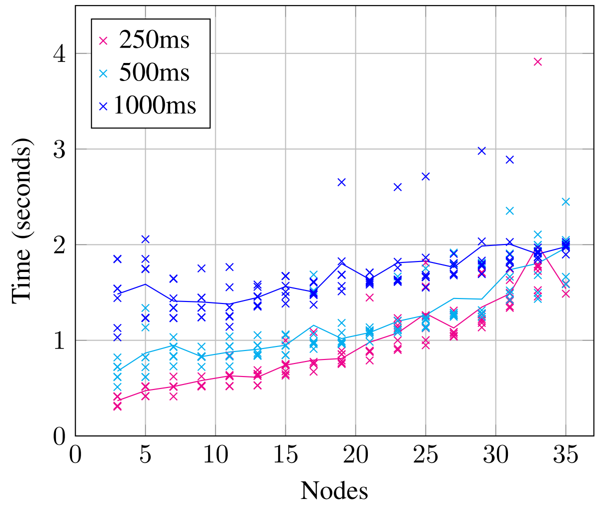

Figure 7. Recovery from Half Cluster Failure

6. Conclusion

The ability to create distributed applications that minimize dependence on centralized infrastructure is important for many use cases. The combination of the Raft distributed consensus algorithm with the WebRTC communication protocol enables a powerful middleware system for building powerful distributed browser-to-browser applications. The Raft over WebRTC middleware system is able to easily scale to at least 35 node clusters and that it can quickly recover from multiple node failures. Further study may show that a dynamically tuned election timer that accounts for communication latency would allow the cluster size to be scaled well beyond 35 nodes.

Acknowledgments

I would like to thank my academic supervisor, David Levine, for his excellent input and oversight of this project. I would also like to thank Diego Ongaro for his amazing work on the Raft protocol and paper which made implementing a Raft implementation a pleasure. There are too many people involved in the creation and standardization of WebRTC but I would like to thank them for their tireless work to make browser-to-browser communication possible. Last but not least, I would like to thank my family for putting up with the long nights and weekends I spent on this project. Thanks!

References

[1] D. Ongaro, “Consensus: Bridging theory and practice,” Ph.D. dissertation, Stanford University, 2014. [Online]. Available: https://github.com/ongardie/dissertation/blob/master/stanford.pdf?raw=true

[2] A. Narayanan, C. Jennings, A. Bergkvist, and D. Burnett, “WebRTC 1.0: Real-time communication between browsers,” W3C, W3C Working Draft, Feb. 2015, http://www.w3.org/TR/2015/WD-webrtc-20150210/.

[3] H. Alvestrand, “Overview: Real time protocols for browser-based applications,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-rtcweb-overview-14, Jun. 2015, http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-overview-14.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-overview-14.txt

[4] J. Rosenberg, “Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols,” RFC 5245 (Proposed Standard), Internet Engineering Task Force, Apr. 2010, updated by RFC 6336. [Online]. Available: http://www.ietf.org/rfc/rfc5245.txt

[5] W. R.-T. C. W. Group. (2015) Web real-time communications working group. [Online]. Available: http://www.w3.org/2011/04/webrtc/

[6] A. Narayanan, C. Jennings, A. Bergkvist, and D. Burnett, “WebRTC 1.0: Real-time communication between browsers,” W3C, W3C Working Draft, Feb. 2015, http://www.w3.org/TR/2015/WD-webrtc-20150210/#rtcpeerconnection-interface.

[7] ——, “WebRTC 1.0: Real-time communication between browsers,” W3C, W3C Working Draft, Feb. 2015, http://www.w3.org/TR/2015/WD-webrtc-20150210/#peer-to-peer-data-api.

[8] E. Rescorla, “Webrtc security architecture,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-rtcweb-security-arch-11, Mar. 2015, http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-arch-11.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-arch-11.txt

[9] ——, “Security considerations for webrtc,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-rtcweb-security-08, Feb. 2015, http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-08.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-08.txt

[10] J. Martin. (2015) Raft.js. [Online]. Available: https://github.com/kanaka/raft.js

[11] J. Guth. (2015, May) docker. [Online]. Available: https://github.com/fentas/docker/blob/a0495cd934703c122db/slimerjs/Dockerfile