Raft Distributed Consensus Algorithm over WebRTC (Web Real-Time Communication)

Joel D. Martin and David Levine

University of Texas at Arlington

September 7, 2015

Abstract

We describe the creation of a browser-based distributed peer-to-peer application using WebRTC, PeerJS and Raft.js. WebRTC is a protocol that modern browsers have begun to integrate which enables browser-to-browser communication. PeerJS is a WebRTC client library and extensible signaling server. Raft.js is an implementation in JavaScript of the Raft distributed consensus protocol. These technologies are combined to create a chat application that guarantees a shared, consistent and linearized view of the chat history without relying on a single centralized server for communication. We also create a series of automated tests using the SlimerJS browser automation tool and use them to show the correctness of the chat application and also to measure the performance and scalability of the application for various cluster sizes.

1 Introduction

In the past few years, browsers have quickly evolved from document rendering engines into powerful platforms for building sophisticated applications. Modern browsers are integrating new protocols for communication including WebSockets for full-duplex browser-to-server communication [8] and WebRTC for direct browser-to-browser communication [26]. These new capabilities enable software developers to treat web applications as first class participants in distributed network applications.

These new browser capabilities also bring with them the traditional challenges of building reliable distributed applications such as: concurrency, security, scalability, naming, discovery, replication, relocation, etc. The solutions to many of these challenges involve the proper management of distributed state. Distributed consensus is one category of distributed state management in which multiple nodes must come to agreement on the state of single value or set of values. The grandparent of distributed consensus algorithms is the Paxos algorithm described by Leslie Lamport [19]. However, Paxos has been notoriously difficult to understand and to implement correctly [6]. Raft is a more recent distributed consensus algorithm that aims for understandability and practicality of implementation [30].

In this paper we will describe an implementation of the Raft protocol in a browser context using WebRTC as the communication channel. We will describe the results of the implementation and look at some of unique challenges (and opportunities) we encountered with the implementation. We will also explore some ideas for further study.

2 Motivation/Use Cases

The traditional use of distributed consensus is in the data-center (private or cloud based) often as the core coordination component of a replicated data-storage system. In these examples, the set of nodes that compose the cluster tend to be fairly static and changes to cluster membership are usually for the purpose of replacing nodes. Changes to the long-term size of the cluster are uncommon.

Most existing distributed browser applications are extensions of the data-centric model in which browsers connect to load-balanced web servers which are in close network proximity to the distributed consensus cluster. The backend web servers deliver the web application to the browser and also serve as the conduit for all browser-to-browser communication and state management.

However, requiring application messages and application state to traverse the back-end servers is not always desirable for various reasons including the following:

1. *security (privacy)*: application state and user messages may need to traverse directly from browser to browser rather than traversing through a server.

2. *locality*: the nature of the application may allow clusters of applications that are nearby on a network (e.g. intranet) to communicate and coordinate directly rather than passing all messages and state through a server back-end which may be considerably more remote.

3. *bandwidth and scale*: a distributed browser based application may be able to avoid expensive bandwidth into and out of the backend data-center by passing messages and state directly between browsers. Depending on the design of the application this may also enable better scalability then a more centralized model.

Here are some specific use cases in which Raft over WebRTC may be applicable:

1. *Consistent order secure chat*: a chat room system where all users see the same message history but the messages never traverse through a central server and there is no central arbiter of trusth. The application code itself may still be delivered from a central web service. This is the use case that was implemented for this paper.

2. *Private multiuser document editing/whiteboard system*: Google Docs, among other examples has demonstrated

a powerful model for collaborative document creation. However, these systems all use a centralized servers to coordinate shared state and handle browser-to-browser communication. Using a Raft over WebRTC could enable applications with similar functionality but without the central coordination and communication.

3. *Browser based network games*: most network games have certain states that must be agreed upon by all nodes (e.g. avatar dead or alive, etc). This can be achieved with Raft over WebRTC without requiring any data to traverse a central server. In addition to reducing browser to server bandwidth requirements, this may also allow lower latency gaming if game instances are partitioned into clusters based on latency.

3 Background

3.1 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" [30].

The Raft algorithm breaks the problem of consensus into three core sub-problems: leader election, log replication, and safety [30, Section 3]. A full Raft system must also address: cluster membership changes [30, Section 4], log compaction [30, Section 5], and client interaction [30, 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 replicated 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 [30, p. 12].

3.1.1 Leader Election

Each node of the Raft cluster maintains an ordinally increasing current term value. When a Raft follower node does not receive a heartbeat from a leader within a random election timeout period, it transitions to candidate state, votes for itself, increases its current term value by one and sends a requestVote RPC (with the new term) to all the other nodes in the cluster. When a node receives a requestVote RPC with a term that is greater than its own it will become a follower (if not already), update its term to the new term, and send a RPC response back that indicates a vote for that candidate. However, a node may only vote for one candidate in a given term. If a candidate receives votes from more than half of the cluster, then it immediately becomes a leader and sends out a appendEntries RPC to confirm leadership and reset the election timers on all the other nodes [30, Section 3.4].

3.1.2 Log Replication

Each Raft node maintains a transaction log. The entries in the transaction log each contain a term value in addition to the action to apply to the state machine. Each node also keep track of the most recent log entry which is known to be committed (exists in the log of more than half the nodes in the cluster).

When a new leader is elected, it begins sending appendEntries RPCs to other nodes in the Raft cluster. These RPC messages contain information about the state of its transaction log: index and term of the latest entry, and the most recently committed entry. If the node does not have an entry matching the most recent entry on the leader, then it replies with false to indicate that its log and not up to date. The next appendEntries RPC that the leader sends to that node will contain the next oldest index and term. This will continue until the leader discovers the latest log entry which is agreement (the node may not have any entries if it is new). Then the leader will begin to propagate entries to the node in subsequent appendEntries RPCs until the node is caught up. The leader continues sending empty appendEntries as heartbeat RPCs to all the nodes until it receives a new entry in its transaction log from a client [30, Section 3.5].

3.1.3 Safety

In order to ensure that each state machine on every node executes exactly the same commands in exactly the same order, the Raft system provides safety guarantees. In particular, there are some restrictions on which Raft nodes are actually eligible to become a leader based on the state of their transaction log compared to other node transaction logs. A Raft node will only vote for a candidate if the candidate has a log that is more up-to-date than the voter. A log entry with a later term is always more up-to-date. If the log entries have the same term, then the entry with a higher index is more up-to-date. In addition, Raft leaders may not consider entries from a previous term to be committed until it has committed at least one entry from its own term [30, Section 3.6].

Raft nodes must also persist certain properties to durable storage before sending or receiving any RPCs. The properties that must be persisted are: current term, current vote for this term (if any), and either the full transaction log or the state machine plus any unapplied transaction log entries [30, Section 3.8].

3.1.4 Membership Changes

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. As soon as the new entry is added to the log it becomes effective on that node without waiting for the entry to be fully committed across the cluster. However, a new cluster change entry may not be added to the log until the previous one is committed. Once the change entry is committed, the initiator of the change

is notified, removed servers can be shut down and another change entry may be added to the log [30, Section 4].

3.1.5 Log Compaction

In order to keep the replicated transaction log from growing indefinitely, Raft nodes should periodically compact their logs. There are many different ways to accomplish log compaction and the best way will depend on application requirements and on the specific nature of the distributed state machine. For example, if the state machine is simply a single shared counter, then any log entries before the most recently committed entry may be safely discarded. 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. Snapshotting the state machine also means that the implementation must provide a way for the leader to serialize and send the current state machine to other nodes if their transaction log is too old (e.g. when a new node is added) [30, Section 5].

3.1.6 Client Interaction

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 either via broadcast, via an external directory service, or directly provided by the user. Once the client discovers a node of the cluster, that node can either forward messages directly to the leader node (if it is not the leader) or it can reject client requests with a response indicating the address of the leader where client requests should be redirected [30, Section 6.2].

The Raft protocol can also provide strict linearizable semantics for client commands/requests (reads and writes) for applications that require it. In order to accomplish this, each client is given a unique ID, and the client assigns a unique serial number to each command. This prevents a single client command from being applied twice (leader crash, network duplication) or lost (packet loss) [30, Section 6.3].

Linearizability is important not just for writes/updates to the state machine, but also for reads, which means that read requests must also be entered as transactions in the log and committed to more than half the Raft nodes before the leader can respond to the client. If reads bypass the transaction log then those reads are serializable but not necessarily linearizable. For example, a leader may have been deposed and not realize it if the cluster is partitioned which could result in reads of stale data (the leader of the larger partition may have already committed new entries). [30, Section 6.4].

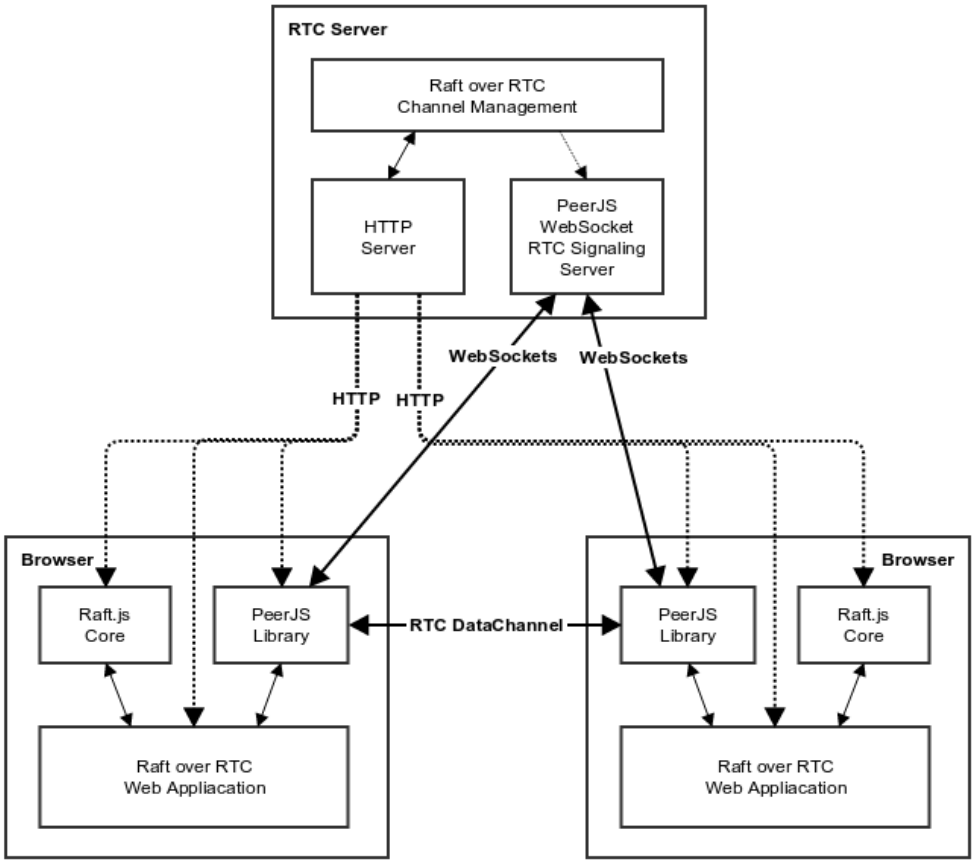


Figure 1: Raft over RTC Architecture

3.2 Raft.js

Raft.js is an implementation of the Raft algorithm in JavaScript that was created by the author of this paper [23]. 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 [30, Consensus: Diego].

3.2.1 Modular Design

The implementation of the Raft algorithm is implemented in the `RaftServerBase` class (in `base.js`). The base class omits the implementation of functions with side-effect such as RPC communication, durable storage, scheduling/timeouts, or state machine command/management. In order to create a working implementation these functions must be provided either as configuration parameters when the class is instantiated or by sub-classing the class.

The modular design of the Raft.js implementation allows it to be easily used in different contexts. For example, the `RaftServerLocal` class (`local.js`) implements RPC calls using plain JavaScript function calls between different instances of the class (nodes) in the same execution context. For example, the online Raft visualization at http://kanaka.github.io/raft.js/ [24] uses the `RaftServerLocal` class and instantiates it using a scheduling function that executes one scheduled event each time the user clicks on the "Step" button rather than due to the passage of time.

The `RaftServerHttp` class (`http.js`) embeds a simple web server into each Raft node and then uses normal HTTP requests to send RPC messages between Raft nodes.

For this paper, the `RaftServerLocal` class was extended with the capability of sending RPCs over the WebRTC Data Channel.

3.2.2 Differences between Raft and Raft.js

The current Raft.js implementation most of the Raft algorithm including: leader election, log replication, safety, membership changes, basic client interaction, and read-only operation optimizations. Log compaction and full client linearizability are not currently implemented (clients and client transactions are not assigned unique IDs). Log compaction is important for long-running production systems in order to limit memory growth and allow new nodes to catch up their logs more efficiently. However log compaction is not required for verifying overall system correctness and does not significantly impact short-running test configurations. Full client linearizability is important feature for simplifying certain types of distributed applications, however the implementation of linearizability requires some significant additions to the Raft implementation (e.g. assigning client IDs and sequence IDs and tracking timed-out clients). For the chat application, simply adding a single atomic check-and-set state machine command is sufficient to achieve a linearized chat history. This does impose a slight additional cost on the client because it must retry the atomic operations until they succeed. The author intends to implement both log compaction and full client linearizability in the future, however these do not have a significant impact on the ability to test Raft combined with WebRTC.

The Raft protocol has undergone some revisions since the original draft paper [31] (including incorporation of suggestions by the author of this paper). These changes include cluster membership simplifications, separating the concepts of committing entries to the log and applying entries to the state machine, clearer client interaction, etc. As part of the work for implementing Raft over WebRTC, the Raft.js implementation was updated to reflect these changes to the Raft protocol definition.

The are also some other differences between Raft.js and Raft as described in [30, Consensus: Diego]:

1. The paper describes the Raft RPCs in terms of synchronous stream based request/response model where requests and responses are automatically correlated together by a lower level of the software stack. For example with HTTP 1.0 [4] client requests are followed by a server response on any given TCP stream. Most remote procedure call mechanims also use a synchronous request/response model. WebRTC DataChannel connections are message based rather than request/response stream based. The original Raft definition of RPC request and response messages almost contain enough information to function correctly in an environment without builtin request/response correlation such as UDP or SCTP when used with reliable transport disabled. Raft.js includes a small modification to the response RPCs for requestVote and appendEntries to add a 'sourceId' field that enables the protocol to function with message based transports.

2. The Raft paper is clear that each membership change must be delayed until all the previous membership change is committed (sequential) [30, Section 4.1]. It is implied that the leader node should track all pending membership change requests and apply them one at a time. However, in Raft.js, if a client requests a membership change while another change request is pending (uncommitted), the leader will reject the change request with a status of 'PENDING_CONFIG_CHANGE'. This keeps the core Raft implementation simpler with the trade-off being that the client implementation of membership change requests is slightly more complex due to retry logic.

3.3 WebRTC

WebRTC is a collection of browser APIs [26] and network protocols [1] that are currently being standardized the W3C and IETF respectively. Collectively these APIs and protocols enable real-time peer-to-peer video, audio and data communication between browsers.

3.3.1 WebRTC Discovery and Signaling

One of the challenges with browser-to-browser communication (and direct peer-to-peer communication in general) is that the web browser environment or user agent (UA) is usually running behind a firewall in a private internet subnet. This complicates direct communication in several ways: the public Internet address of the user agent is unknown and dynamic, inbound connections are denied by default,

and outbound connections are network address translated (NATed) [2, Section 3.4].

To establish a direct communication channel between browsers, WebRTC uses mechanism called Interactive Connectivity Establishment (ICE) that is similar to the Session Initiation Protocol (SIP) commonly used for Voice-over-IP (VoIP) communication [36]. This involves a third party server known as a signaling server. Each user agent (browser) connects to this server at a well known address in order to establish a direct connection to other user agents. The signaling server is also uses to communicate session control messages (open, close, error) and to negotiate media capabilities. 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.

3.3.2 WebRTC APIs

The WebRTC browser APIs enable web applications written in JavaScript 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) [11].

There are three primary WebRTC interfaces:

1. *RTCPeerConnection* [27]: The RTCPeerConnection API represents a direct connection between two browsers. Each RTCPeerConnection object has an associated Interactive Connectivity Establishment (ICE) agent that is used to establish network connectivity through firewalls and NAT services.

2. *MediaStream (aka getUserMedia)* [28]: The MediaStream API represents a stream of audio or video data. MediaStream data can easily be sent and received over RTCPeerConnections. The MediaStream API definition provides a mechanism that allows JavaScript to enumerate the native MediaStreams that are provided by the browser such as a local web camera or a live video stream of the desktop (for desktop sharing). Although support for MediaStreams was the original impetus for the creation of the WebRTC APIs and protocols, the Raft over WebRTC implementation does not make use of the MediaStream API.

3. *RTCDataChannel* [29]: The RTCDataChannel API provides a mechanism for sending and receiving generic data between User Agents (browsers). A single RTCPeerConnection can contain multiple RTCDataChannels. The RTCDataChannel API is modeled after the WebSocket API, however, unlike the WebSocket API which is reliable and guaranteed order transport (like TCP), each RTCDataChannel can be configured with different reliability and ordering settings. For example, by relaxing the reliability and ordering constraints on a RTCDataChannel, the average latency and jitter per message can be decreased even though some messages may be dropped or arrive out of order. The Raft protocol is specifically designed to support dropped or out-of-order messages so the transport options available with RTCDataChannel can be used to gain extra efficiency.

3.3.3 WebRTC Protocol

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. 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:

1. data transport
2. data framing and security
3. connection management
4. data formats
5. presentation and control
6. and local system support functions

The last three functionality groups focus on interactive audio and video applications and are less relevant to Raft over WebRTC. The WebRTC section of the bibliography has a (incomplete) list of WebRTC specific protocols drafts and other related protocol standards for the first three functionality groups (data transport, data framing and security, and connection management).

The signaling transport itself is not defined as part of WebRTC. This is up to the application designer. The JavaScript interface used for establishing the initial connection (signaling) in the WebRTC 1.0 standard is somewhat complicated. A competing standard called Object Real-time Communication (ORTC) is working to define a simpler and more modular JavaScript API for WebRTC signaling [10]. However, at the time of this paper, the ORTC interface was not yet widely supported and still in a state of flux. The use of PeerJS provides an abstraction that is likely to transparently support ORTC in the future.

3.4 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.

3.4.1 PeerJS Server

The PeerJS project provides a simple WebRTC signaling server called PeerServer. This server uses HTTP and WebSockets transport protocols to perform WebRTC signaling on behalf of browser clients. The PeerServer server is also extensible so that more advanced services can be built with it. The PeerJS project includes an extension to PeerServer called ExpressPeerServer that combines a PeerJS signaling server with the popular Node.js ExpressJS web application framework.

3.4.2 PeerJS Client Library

The PeerJS client library provides an simple abstraction over the native browser WebRTC APIs. The WebRTC API standards are not yet finalized and so different browsers and browser releases may support different versions of the WebRTC draft APIs. The PeerJS client library abstracts over these differences and provides a common interface so that the application developer does not have to deal directly with browser differences.

The PeerJS client library also provides a default signaling mechanism that is designed to operate with a PeerJS signaling server. This is particularly helpful because much of the WebRTC signaling transport and protocol is not defined as part of the WebRTC standardization effort. The PeerJS organization provides a cloud-hosted version of a PeerJS compatible signaling server, however an application developer can also run their own signaling service using PeerJS PeerServer.

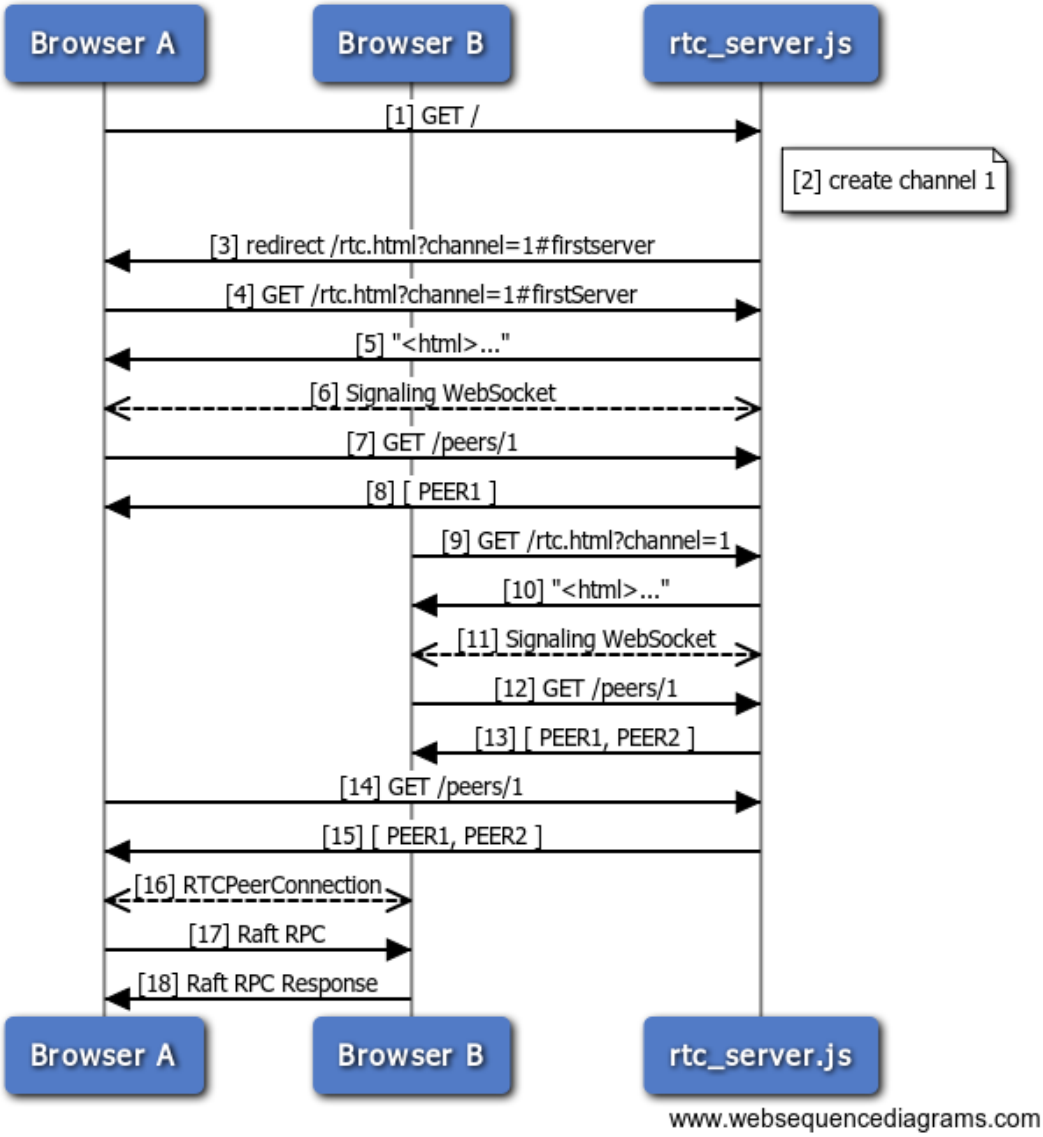


Figure 2: Raft over RTC Sequence Diagram

4 Design and Implementation

4.1 Incremental Steps

During the planning phase, the project was broken down into ten incremental steps:

1. Simple WebRTC messaging: implement a small JavaScript application that uses the PeerJS client library and an unmodified PeerJS signaling server. Demonstrate communication over WebRTC DataChannel using two browser instances (or browser tabs).

2. Raft.js over WebRTC: combine the existing Raft.js implementation with the small JavaScript application

above to demonstrate Raft over WebRTC. This step uses a statically configured three node cluster (three browser tabs). The PeerJS signaling server is polled to retrieve the list of other clients and the Raft cluster is started when the number of PeerJS clients reaches three.

3. Update Raft.js protocol: update the Raft.js implementation to match the latest description of the Raft protocol [30]. This includes the implementation of the simplified membership change algorithm (sequential changes) and the separation of the log commit and state machine apply concepts. Additionally, the new Raft.js implementation always starts a new Raft cluster as a cluster with a single node and additional nodes are dynamically added to the cluster [30, Section 4.4].

4. Dynamic Raft.js over WebRTC: enhance the signaling server with support for multiple sessions/channels to enable separate Raft clusters with the same signaling server. Start each cluster with a single node and automatically add and remove Raft nodes as they come and go.

5. Visual representation: display Raft node state and RPC statistics in addition to the internal Raft logging messages.

6. Sessionless/Datagram RPC messages: change Raft RPC requests and response messages so that RPC responses have enough context to be handled separately from RPC requests. Specifically this enables Raft to support datagram (sessionless) network transports like WebRTC.

7. Atomic state machine operation: implement a state machine update operation that only succeeds if no other operation has been applied to the same key since the last time this client read from that key.

8. Automatic client request forwarding: extend the client request RPC to also be sessionless/datagram based. Implement an asynchronous JavaScript function that determines which node is the leader and forwards client request RPCs to the leader. This function must also correlate RPC responses to the client request and invoke the callback for the original request.

9. Chat application: add the ability to send a chat message to the log using the client request forwarding functionality together with atomic state machine operation. Show the current chat history based on the chat entries in the state machine.

10. Server as Raft.js peer (aspirational): enable server nodes (Raft.js running on Node.js) to fully participate as Raft cluster nodes.

Steps 1-9 were successfully implemented for this paper. Step 10 was an aspirational goal because this functionality is not yet supported in PeerJS [18].

4.2 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 signaling function, the server also provides the following service endpoints:

1. Static web service endpoint: this serves the static HTML, CSS and JavaScript that make up the Raft over WebRTC web application.

2. New channel endpoint: when a browser loads this endpoint, a new PeerJS channel is created and the browser is redirected to a URL for loading the main application (via static file service above). The redirect URL has a query string appended that contains the channel ID of the new channel that was just created. In addition the redirect URL has a fragment identifier (hash/history) that communicates to the web application that this is the first server in the channel.

3. Peer list endpoint: this endpoint takes a channel ID and returns the list of WebRTC peers (browser user agents) that are currently connected to that channel.

Refer to Section 4.3 and Figure 2 for a description of how these endpoints interact.

4.3 Client

The Raft over WebRTC testing application is designed as a single web page (`rtc.html` or `chat.html`). The application has different starting behavior depending on the value of the URL query string (the URL component following the "?" and up to the "#") and the fragment identifier (the URL component after the "#").

When a new Raft cluster is created, the user agent (browser) connects to the 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 an indication that this is the very first Raft cluster node in the fragment identifier (Figure 2 [3]).

The browser follows the redirect and loads the static web application at the static file endpoint provided by the server (Figure 2 [4]). This includes the main application page (`rtc.html` or `chat.html`), the core raft algorithm (`base.js` and `local.js`), the PeerJS client library (`peer.js`), the actual Raft over WebRTC implementation (`rtc.js`), a layout stylesheet (`web/demo.css`), and higher-level application code if that is being loaded (`chat.js`) (Figure 2 [5]).

If the URL fragment identifier is set to "firstServer", then this Raft node is initialized as the leader of a single node cluster and all normal timers are started. The fragment identifier is then unset. The query parameter contains the ID of this cluster (the PeerJS channel).

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 (see 3).

When each page first loads, a PeerJS signaling connection is established with the server (Figure 2 [6] and [11]). Once this connection is established, the application will be notified when any PeerJS clients connect to or disconnect from the server. After the PeerJS signaling connection is established, the client immediately queries the Peer list endpoint (Figure 2 [7] and [12] and [14]]) to discover any clients that were connected prior to when this node connected to the server (Figure 2 [8] and [13] and [15]]).

The web application also begins a periodic async polling function (`addRemoveServersAsync`) when the page loads. When the node is a Raft leader, the function 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 [30, 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.

In addition to the low level Raft message log, the following statistics are shown in the application:

1. *Term*: the current Raft leadership term.

2. *State*: the current Raft node state (follower, candidate, or leader)

3. *Cluster Size*: a count of the current Raft cluster membership (number of nodes currently participating in the cluster)

4. *Log Length*: number of entries in the transaction log.

5. *Request Votes Sent/Received*: the number of 'requestVote' RPCs that have been sent from and received by this node.

6. *Append Entries Sent/Received*: the number of 'appendEntries' RPCs that have been sent from and received by this node.

4.3.1 Client Request Redirects

In the Raft over WebRqTC implementation each local browser is a client of the Raft cluster. However in Raft, the leader node is the only node that adds entries to the log so clients on follower nodes must have a way to communicate with the leader. This is implemented as a pair of functions named `clientRequest` and `clientRequestResponse`. The `clientRequest` function is called by the client with the arguments of a client request RPC and a callback function to be called once the request is completed. The `clientRequest`/`clientRequestResponse` functions work as follows:

1. the `clientRequestResponse` is registered with the local Raft.js implementation as the callback function anytime a `clientRequestResponse` RPC is received.

2. the `clientRequest` function keeps track of the current leader ID. If the current leader ID is not set or this node is the leader then the client request RPC is sent directly to the local Raft node bypassing the network transport layer. If the current leader ID is set then the request is sent to that node as a normal RPC request over the WebRTC channel.

3. when the `clientRequestResponse` function is called and the status is "NOT_LEADER" then the current leader ID being tracked by the `clientRequest` function is set to the leader ID specified in the response. Then the original request is re-issued over the normal WebRTC channel to the leader.

4. when the `clientRequestResponse` function is called with a successful status, the original callback argument of the `clientRequest` function is called with the results from the RPC response.

4.4 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 (see Figure 5). 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. A function to flush sends runs periodically (currently every 100ms) and does the following:

1. If there is line of text that has been sent but not yet added (applied) to the state machine or there are no lines of text on the pending send queue, then the function does nothing.

2. A text line is removed from the head of the pending send queue and stored in *curSend* to indicate that a send is in progress. Then a client request is made using an atomic *seqAppend* command to add the new text to the state machine (*seqAppend* is describe below). The *seqAppend* operation attempts to append the text line to a list at the *history* key in the state machine.

3. When the atomic *seqAppend* operation completes, it may have failed (if another text line was appended first), in which case the flush function puts the failed text line back on the pending send queue. Whether the send was successful or failed, the *curSend* variable is set to nil/null to indicate that no send is in progress. The next time the flush function runs it will attempt to send again.

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 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.

5 Results

5.1 Manual Testing

5.1.1 Setup

The initial way that the application was tested was by loading the basic application to create a Raft over WebRTC cluster with a single node acting as leader. Figure 3 shows a screenshot of the browser after the first node is created:

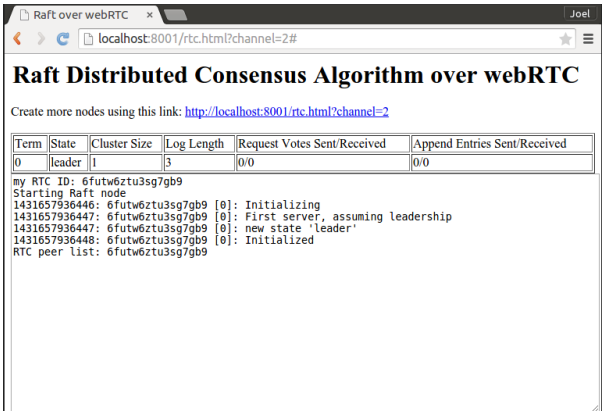


Figure 3: Basic app with one Node/Tab

Then four more Raft nodes were then added to the cluster by clicking on the embedded "create more nodes" link in the application. Figure 4 shows a screenshot of the browser once four new nodes are added to the cluster (5 total):

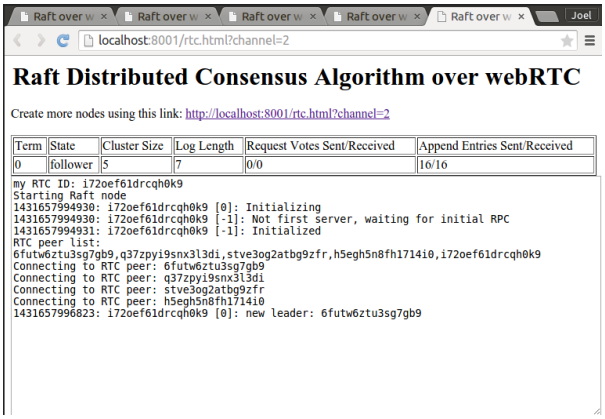


Figure 4: Basic app with Five Nodes/Tabs

In addition, the full chat application was also tested manually by loading the initial chat application landing page to create a Raft over WebRTC cluster with a single leader acting as leader. Then lines of text were entered into the text area each followed by clicking the Send button. Figure 5 shows a screenshot of the browser after the first node/tab of the chat application is created and a line of text was sent:

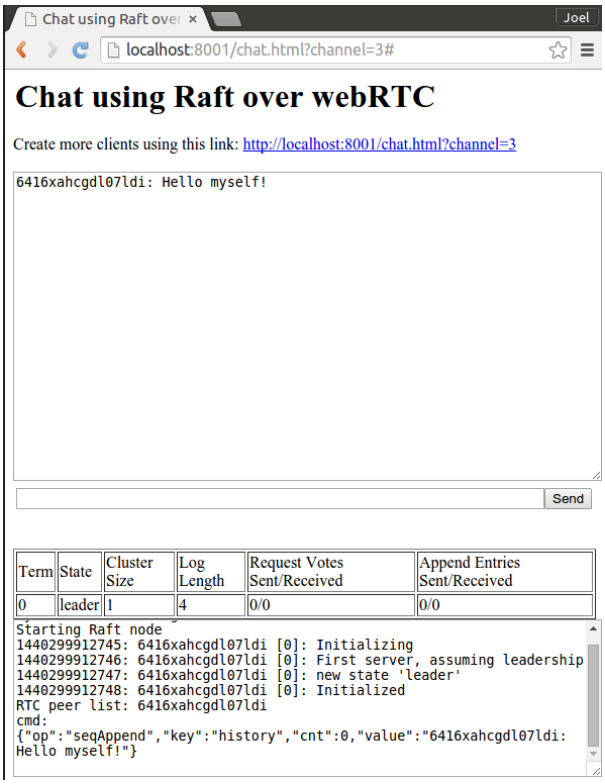


Figure 5: Chat app with one Node/Tab

Then four more Raft nodes were then added to the cluster by clicking on the embedded "create more clients" link in the application. Finally, on a non-leader node more text lines were entered and sent by clicking the Send button. Figure 6 shows a screenshot of the browser once four new nodes/tabs are added to the cluster (5 total) and another text line was sent from the fifth node:

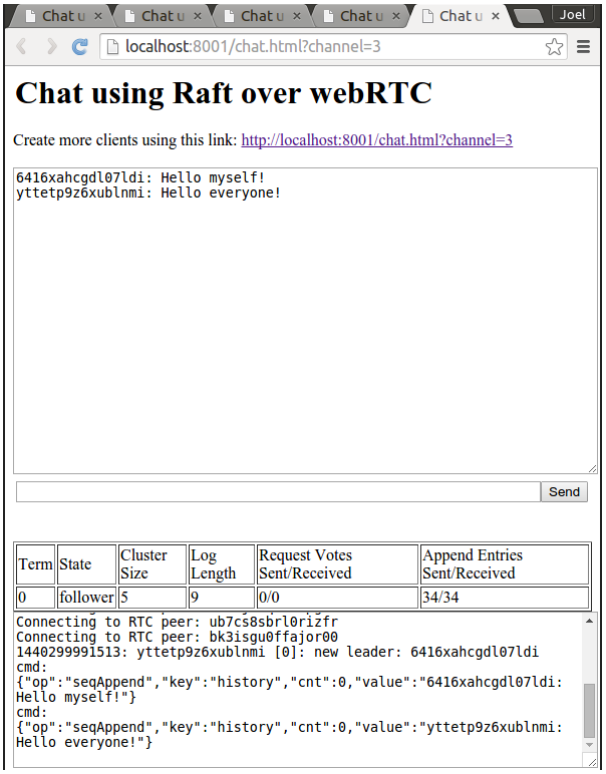


Figure 6: Chat app with Five Nodes/Tabs

After every change was made to the cluster all the nodes were checked to ensure that the following steady state properties were observed:

1. only a single leader
2. all nodes have the same term and transaction log count
3. all non-leader nodes are receiving 'appendEntries' RPCs from the leader
4. all nodes have the same cluster membership state
5. the transaction log entries are identical on all nodes
6. the state machine is identical on all nodes

The current leader was then terminated. The other nodes were checked again the steady state properties. This was repeated until only two nodes remained in the cluster.

Next the five node cluster was brought up again and state machine commands were sent to leader to add to the transaction log. Again the nodes were checked for the above steady state properties.

Then the application was tested was by loading the initial cluster creation landing page in a browser tab. Nodes were randomly added to the cluster (by clicking on the link embedded in the page) or removed from the cluster (by closing a tab). After each change the steady state properties were checked.

Similar testing to the above was done with browser instances on two separate computers. In addition, cross-browser functionality was verified by running nodes from the same cluster running simultaneously in Chrome 42 and Firefox 37.

5.1.2 Outcome

The results of the manual testing above showed that Raft over WebRTC was able to perform leader elections correctly, the log and state machine was replicated correctly to all nodes, and the safety properties were maintained. During testing the membership of the cluster was changed numerous times (both node additions and removals) and the Raft properties were maintained throughout. In addition, client interaction was tested to make sure commands were still properly added to the state machine and transaction log.

One limitation of the Raft protocol is that the if the cluster membership drops from two nodes to one node then the cluster will become unavailable without manual intervention. This is because when there are two nodes in the cluster, both nodes are required for for a majority vote and when one member is forcible shutdown then the remaining cluster node will be unable to win elections or even commit transaction entries and the cluster will become unavailable. There are several options to get around this problem:

1. The node can ask the leader to be removed and then wait for the transaction to become effective before it disconnects. This may not always be an option especially in a browser context where the user may immediately kill the window/tab without giving the application a chance to remove itself from the cluster.
2. The signaling server can serve as a tie-breaker. This would allow the remaining leader successfully commit the removal of the other node. In the case of a partition, the signaling server must only vote for one of the nodes.
3. The node with the lower ID could have an implicit tie-breaker vote. This would allow one node of the partition to continue in the case of a network partition. If a node dies or is killed, this would only allow the cluster to continue to operate if the node that survives happens to have the lower ID.
4. The system could allow manual user intervention. However, this is only likely to work correctly in the case where the user of the last remaining node can be certain that other nodes will never return otherwise there is a risk of data-corruption.

Running a Raft cluster in a browser context can lead to some level of timer instability. The JavaScript execution context is a fully event driven environment with a single thread of execution for a given JavaScript context. JavaScript code is run when browser (or Node.js) events trigger JavaScript code that is registered to that event. In order to implement logic that runs periodically, JavaScript code is registered as a callback timer using the setTimer, setInterval or requestAnimationFrame APIs. However, none of these mechanisms is a precise timer. When a timer fires, the callback code they refer to is added to the JavaScript execution queue and events on the queue are processed in the order they arrive. This means callbacks may be delayed by other code that is already running or queued first [35].

However, there is another browser mechanism that is less well known but that causes even larger delays in timer code execution. Modern browsers often have dozens or even

hundreds of windows and/or tabs open simultaneously. In order to reduce CPU load, increase user responsiveness and improve battery life, most modern browsers will throttle the JavaScript callback timers for all windows and tabs that are hidden or have not been directly interacted with for some period of time. This throttling can cause timers to slow down by an order of magnitude or more.

In the context of Raft over WebRTC, the slowdown of background page timers means that whenever the leader node is residing on a page that is in the background, the propagation of transaction log entries slows down substantially. This also means that other nodes which are in the foreground are more likely to have an election timeout timer fire and they are more likely to become a leader because they will be able to send and receive votes more quickly than nodes that are in the background. In this sense the problem is self-correcting because over the long term because higher performing nodes more are likely to start and win elections. This also indicates dynamic adjustment of timeout values may be a fruitful area of further study.

In the context of the full chat application the slowdown of background pages is even more pronounced because each addition to the chat history can take several messages to fully propagate the new date. An interesting area of further study would be to extend the Raft protocol to more proactively adapt to network and node performance conditions (delayed timers) and more intentionally transition leadership to more capable nodes.

5.2 Automated Testing

5.2.1 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 a 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 was used to encapsulate and execute the SlimerJS environment. The container is built using *test/Dockerfile* which is in turn derived from the "fentas/slimerjs" docker image [12]. 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 testing.

Writing tests using SlimerJS involves creating JavaScript programs that are run within the SlimerJS environment. These scripts can create browser instances (basically headless browser tabs), query the status of those pages and execute JavaScript within the context of the pages themselves.

Three different tests were created to test the basic Raft over WebRTC application and the chat application. Each of the tests takes a parameter indicating the number of nodes/pages to launch and test again. The tests scripts 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 recover from the failure.

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 the following configuration:

1. *CPUs*: 8 Intel Core i7 processors running at 3.20GHz

2. *Memory*: 12 GB

3. *OS*: Ubuntu 14.04.1 LTS

4. *Docker*: version 1.3.1

5. *Docker image*:

   (a) *OS*: Ubuntu 12.04.5 LTS

   (b) *SlimerJS*: version 0.9.6

   (c) *XULRunner*: 38.0.5

Three tests were run at 10 different cluster sizes (odd cluster sizes from 3 to 21 nodes): *test/test_up.js* and *test/test_kill_nodes.js* in kill one (leader) node and kill half nodes. Each of these tests was run with three different Raft election timeout values: 100ms, 500ms, 1000ms. The *test/test_chat_propagate.js* test was run at 11 different cluster sizes (odd cluster sizes from 1 to 21 nodes). Each test mode and cluster size was run 7 times for a total of 707 test runs (3*10*7*3 + 11*7)

5.2.2 Outcome

Figure 7 shows the results of running the *test/test_chat_propagate.js* test to show the amount of time that a stable cluster takes to propagate, commit and apply a single state machine command to the state machines on every cluster node.

Figure 7 is a scatterplot with each test run as a blue "x" and a black line representing the average at each cluster size.

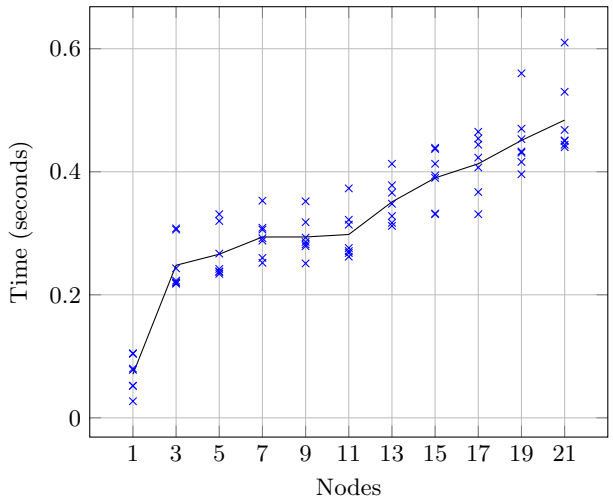


Figure 7: Propagate, Commit, and Apply Command

Figures 8, 9 and 10 show the results of running the *test/test_up.js* and *test/test_kill_nodes.js* in both modes (kill only leader and kill just under half the nodes). Three different scatterplots are shown for each test mode to test different values of the Raft election timeout value: 100ms, 500ms, and 1000ms. The election timeout value makes very little difference to the average time curve line. The reason for this is that the first leader tends to stay the leader as new nodes are added so for the most part the election timer does not have an impact.

Some specific notes about the figures:

1. *Figure 8 (a)*: The results show that once the cluster size reaches 17, the cluster sometimes does not activate before the test times out (the red "x" marks). Once the cluster reaches 21 nodes, all 7 test runs time out. The reason for this is because at 17 nodes, the leader is not always able to send out appendEntries RPCs to all the other nodes before their election timeout fires. More analysis is needed, but one possible explanation is that the single SlimerJS XULRunner process that is handling all the cluster nodes is not able to give enough CPU time to the leader node to reliable send appendEntries RPCs to all the other nodes within the election timeout value. Increasing the election timeout to 500ms or greater eliminates the failures without significantly changing the average cluster activation time curve.

2. *Figure 9 (a)*: The results show that when the cluster size reaches 15, then average amount of time for cluster recovery (after the leader is killed) increases dramatically (not all data points are shown on the plot for size 15). At cluster size 17 the cluster is no longer able to recover within the test timeout period. Increasing the election timeout to 500ms or greater eliminates the failures at the cost of slightly increased recovery time for smaller cluster sizes.

3. *Figure 10 (a)*: During this test cluster was unable to activate at cluster size 21 when the election timeout value is set to 100ms (see Figure 8 (a)). This means there was no opportunity for cluster recovery at cluster size 21 so there is no data for that column.

Each figure is a scatterplot with each test run as a blue "x" and a black line representing the average at each cluster size.

6 Next Steps

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 the use of 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.

2. Test the survivability of the cluster when the signaling server becomes completely unresponsive or very slow.

3. Explore models for dynamically adjusting Raft timeout values to account for changing network and browser conditions.

4. Explore models for dealing with the case where the cluster goes from two nodes down to one node. Can usefulness and availability be improved without sacrificing Raft safety constraints.

5. Test the system in more diverse network environments. Determine the reliability of establishing direct browser-to-browser communication. Determine if corporate firewalls make this as difficult as is implied in some of the documentation and standards texts.

6. Perform quantitative and real-world testing to better characterize the performance, availability and scalability characteristics of the system.

7. 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.

8. 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.

9. Test larger scale deployments spanning larger and more diverse network domains. Determine the factors that affect scalability and characterize scaling limits.

10. Perform wider cross-browser testing including Internet Explorer and Opera.

11. Implement Raft log compaction and full client linearizability.

12. Explore models that allow for a combination of strict shared state and high performance bulk data such as media or object blobs. For example, the data with full consensus might contain hashes that to the refer to bulk data.

13. Implement a Node.js based Raft over WebRTC application. This would allow non-browser contexts to participate in the Raft cluster together with the browser clients. PeerJS issue #103 must be resolved for this to be feasible [18].

14. Explore a design where some nodes of the cluster are non-voting and are not counted as part of transaction commit quorum. However, these nodes would still be included in the log replication algorithm. One model this would enable would be a configuration where a small group of voting nodes serve as the hub for a larger group of non-voting nodes. This may enable higher scaling of the system. It also enables interesting applications such as a logging or backup 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 [20, p. 215] which made implementing a Raft implementation a pleasure [20]. 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!

General references

[4] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945 (Informational). Internet Engineering Task Force, May 1996. URL: http://www.ietf.org/rfc/rfc1945.txt.

[6] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*. Ed. by Indranil Gupta and Roger Wattenhofer. ACM, 2007, pp. 398–407. ISBN: 978-1-59593-616-5. DOI: 10.1145/1281100.1281103. URL: http://doi.acm.org/10.1145/1281100.1281103.

[8] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455 (Proposed Standard). Internet Engineering Task Force, Dec. 2011. URL: http://www.ietf.org/rfc/rfc6455.txt.

[9] J. Fischl, H. Tschofenig, and E. Rescorla. *Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)*. RFC 5763. RFC Editor, May 2010.

[10] Object-RTC API Community Group. *Object RTC (ORTC) API for WebRTC*. Aug. 2014. URL: http://ortc.org/wp-content/uploads/2014/08/ortc.html (visited on 09/06/2015).

[11] Web Real-Time Communications Working Group. *Web Real-Time Communications Working Group*. 2015. URL: http://www.w3.org/2011/04/webrtc/ (visited on 09/06/2015).

[12] Jan Guth. *docker. Collection of Dockerfiles*. May 2015. URL: https://github.com/fentas/docker/blob/a0495cd934703c122db/slimerjs/Dockerfile (visited on 09/07/2015).

[13] Mark Handley and Van Jacobson. *SDP: Session Description Protocol*. RFC 2327. http://www.rfc-editor.org/rfc/rfc2327.txt. RFC Editor, Apr. 1998. URL: http://www.rfc-editor.org/rfc/rfc2327.txt.

[15] Emil Ivov, Eric Rescorla, and Justin Uberti. *Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol*. Internet-Draft draft-ietf-mmusic-trickle-ice-02. http://www.ietf.org/internet-drafts/draft-ietf-mmusic-trickle-ice-02.txt. IETF Secretariat, Jan. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-mmusic-trickle-ice-02.txt.

[18] Jos de Jong. *PeerJS issue 103*. 2013. URL: https://github.com/peers/peerjs/issues/103 (visited on 09/07/2015).

[19] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[20] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[23] Joel Martin. *Raft.js. Raft.js is an implementation of the Raft consensus algorithm in JavaScript*. 2015. URL: https://github.com/kanaka/raft.js (visited on 09/05/2015).

[24] Joel Martin. *Raft.js Visualization*. 2013. URL: http://kanaka.github.io/raft.js/ (visited on 09/05/2015).

[26] Anant Narayanan et al. *WebRTC 1.0: Real-time Communication Between Browsers*. W3C Working Draft. http://www.w3.org/TR/2015/webrtc-20150210/. W3C, Feb. 2015.

[27] Anant Narayanan et al. *WebRTC 1.0: Real-time Communication Between Browsers. RTCPeerConnection Interface*. W3C Working Draft. http://www.w3.org/TR/2015/WD-webrtc-20150210/#rtcpeerconnection-interface. W3C, Feb. 2015.

[28] Anant Narayanan et al. *WebRTC 1.0: Real-time Communication Between Browsers. Media Stream API Extensions for Network Use*. W3C Working Draft. http://www.w3.org/TR/2015/WD-webrtc-20150210/#media-stream-api-extensions-for-network-use. W3C, Feb. 2015.

[29] Anant Narayanan et al. *WebRTC 1.0: Real-time Communication Between Browsers. Peer-to-peer Data API*. W3C Working Draft. http://www.w3.org/TR/2015/WD-webrtc-20150210/#peer-to-peer-data-api. W3C, Feb. 2015.

[30] Diego Ongaro. "Consensus: Bridging Theory and Practice". PhD thesis. Stanford University, 2014. URL: https://github.com/ongardie/dissertation/blob/master/stanford.pdf?raw=true.

[31] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[35] J. Resig and B. Bibeault. *Secrets of the JavaScript Ninja*. Manning Pubs Co Series. Manning, 2013. ISBN: 9781933988696. URL: https://books.google.com/books?id=ab8CPgAACAAJ.

[37] J. Rosenberg and H. Schulzrinne. *An Offer/Answer Model with Session Description Protocol (SDP)*. RFC 3264. http://www.rfc-editor.org/rfc/rfc3264.txt. RFC Editor, June 2002. URL: http://www.rfc-editor.org/rfc/rfc3264.txt.

[48] Justin Uberti, Cullen Jennings, and Eric Rescorla. *Javascript Session Establishment Protocol*. Internet-Draft draft-ietf-rtcweb-jsep-10. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-jsep-10.txt. IETF Secretariat, June 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-jsep-10.txt.

WebRTC general references

[1] Harald Alvestrand. *Overview: Real Time Protocols for Browser-based Applications*. Internet-Draft draft-ietf-rtcweb-overview-14. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-overview-14.txt. IETF Secretariat, June 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-overview-14.txt.

[33] Eric Rescorla. *Security Considerations for WebRTC*. Internet-Draft draft-ietf-rtcweb-security-08. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-08.txt. IETF Secretariat, Feb. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-08.txt.

[34] Eric Rescorla. *WebRTC Security Architecture*. Internet-Draft draft-ietf-rtcweb-security-arch-11. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-arch-11.txt. IETF Secretariat, Mar. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-arch-11.txt.

WebRTC data transport

[2] Harald Alvestrand. *Transports for WebRTC*. Internet-Draft draft-ietf-rtcweb-transports-08. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-transports-08.txt. IETF Secretariat, Feb. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-transports-08.txt.

[5] G. Camarillo, O. Novo, and S. Perreault. *Traversal Using Relays around NAT (TURN) Extension for IPv6*. RFC 6156. RFC Editor, Apr. 2011.

[7] Subha Dhesikan et al. *DSCP and other packet markings for RTCWeb QoS*. Internet-Draft draft-ietf-tsvwg-rtcweb-qos-03. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-rtcweb-qos-03.txt. IETF Secretariat, Nov. 2014. URL: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-rtcweb-qos-03.txt.

[14] Christer Holmberg, Salvatore Loreto, and Gonzalo Camarillo. *Stream Control Transmission Protocol (SCTP)-Based Media Transport in the Session Description Protocol (SDP)*. Internet-Draft draft-ietf-mmusic-sctp-sdp-14. http://www.ietf.org/internet-drafts/draft-ietf-mmusic-sctp-sdp-14.txt. IETF Secretariat, Mar. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-mmusic-sctp-sdp-14.txt.

[16] Randell Jesup, Salvatore Loreto, and Michael Tuexen. *WebRTC Data Channel Establishment Protocol*. Internet-Draft draft-ietf-rtcweb-data-protocol-09. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-protocol-09.txt. IETF Secretariat, Jan. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-protocol-09.txt.

[17] Randell Jesup, Salvatore Loreto, and Michael Tuexen. *WebRTC Data Channels*. Internet-Draft draft-ietf-rtcweb-data-channel-13. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-channel-13.txt. IETF Secretariat, Jan. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-channel-13.txt.

[21] J. Lazzaro. *Framing Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP) Packets over Connection-Oriented Transport*. RFC 4571. http://www.rfc-editor.org/rfc/rfc4571.txt. RFC Editor, July 2006. URL: http://www.rfc-editor.org/rfc/rfc4571.txt.

[22] R. Mahy, P. Matthews, and J. Rosenberg. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 5766. http://www.rfc-editor.org/rfc/rfc5766.txt. RFC Editor, Apr. 2010. URL: http://www.rfc-editor.org/rfc/rfc5766.txt.

[25] D. McGrew and E. Rescorla. *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)*. RFC 5764. http://www.rfc-editor.org/rfc/rfc5764.txt. RFC Editor, May 2010. URL: http://www.rfc-editor.org/rfc/rfc5764.txt.

[32] Colin Perkins, Magnus Westerlund, and Joerg Ott. *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP*. Internet-Draft draft-ietf-rtcweb-rtp-usage-25. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-rtp-usage-25.txt. IETF Secretariat, June 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-rtp-usage-25.txt.

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

[38] J. Rosenberg et al. *Session Traversal Utilities for NAT (STUN)*. RFC 5389. http://www.rfc-editor.org/rfc/rfc5389.txt. RFC Editor, Oct. 2008. URL: http://www.rfc-editor.org/rfc/rfc5389.txt.

[39] J. Rosenberg et al. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. RFC 3489. http://www.rfc-editor.org/rfc/rfc3489.txt. RFC Editor, Mar. 2003. URL: http://www.rfc-editor.org/rfc/rfc3489.txt.

[40] J. Rosenberg et al. *TCP Candidates with Interactive Connectivity Establishment (ICE)*. RFC 6544. http://www.rfc-editor.org/rfc/rfc6544.txt. RFC Editor, Mar. 2012. URL: http://www.rfc-editor.org/rfc/rfc6544.txt.

[42] P. Srisuresh, B. Ford, and D. Kegel. *State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)*. RFC 5128. http://www.rfc-editor.org/rfc/rfc5128.txt. RFC Editor, Mar. 2008. URL: http://www.rfc-editor.org/rfc/rfc5128.txt.

[43] R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. http://www.rfc-editor.org/rfc/rfc4960.txt. RFC Editor, Sept. 2007. URL: http://www.rfc-editor.org/rfc/rfc4960.txt.

[44] R. Stewart et al. *Stream Control Transmission Protocol (SCTP) Partial Reliability Extension*. RFC 3758. http://www.rfc-editor.org/rfc/rfc3758.txt. RFC Editor, May 2004. URL: http://www.rfc-editor.org/rfc/rfc3758.txt.

[45] Randall Stewart et al. *A New Data Chunk for Stream Control Transmission Protocol*. Internet-Draft draft-stewart-tsvwg-sctp-ndata-03. http://www.ietf.org/internet-drafts/draft-stewart-tsvwg-sctp-ndata-03.txt. IETF Secretariat, Oct. 2013. URL: http://www.ietf.org/internet-drafts/draft-stewart-tsvwg-sctp-ndata-03.txt.

[46] Martin Thomson. *Application Layer Protocol Negotiation for Web Real-Time Communications (WebRTC)*. Internet-Draft draft-ietf-rtcweb-alpn-01. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-alpn-01.txt. IETF Secretariat, Feb. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-alpn-01.txt.

[47] Michael Tuexen et al. *DTLS Encapsulation of SCTP Packets*. Internet-Draft draft-ietf-tsvwg-sctp-dtls-encaps-09. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-sctp-dtls-encaps-09.txt. IETF Secretariat, Jan. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-sctp-dtls-encaps-09.txt.

WebRTC framing and security

[3] M. Baugher et al. *The Secure Real-time Transport Protocol (SRTP)*. RFC 3711. http://www.rfc-editor.org/rfc/rfc3711.txt. RFC Editor, Mar. 2004. URL: http://www.rfc-editor.org/rfc/rfc3711.txt.

[16] Randell Jesup, Salvatore Loreto, and Michael Tuexen. *WebRTC Data Channel Establishment Protocol*. Internet-Draft draft-ietf-rtcweb-data-protocol-09. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-protocol-09.txt. IETF Secretariat, Jan. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-protocol-09.txt.

[17] Randell Jesup, Salvatore Loreto, and Michael Tuexen. *WebRTC Data Channels*. Internet-Draft draft-ietf-rtcweb-data-channel-13. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-channel-13.txt. IETF Secretariat, Jan. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-channel-13.txt.

[32] Colin Perkins, Magnus Westerlund, and Joerg Ott. *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP*. Internet-Draft draft-ietf-rtcweb-rtp-usage-25. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-rtp-usage-25.txt. IETF Secretariat, June 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-rtp-usage-25.txt.

[41] H. Schulzrinne et al. *RTP: A Transport Protocol for Real-Time Applications*. STD 64. http://www.rfc-editor.org/rfc/rfc3550.txt. RFC Editor, July 2003. URL: http://www.rfc-editor.org/rfc/rfc3550.txt.

WebRTC connection management

[9] J. Fischl, H. Tschofenig, and E. Rescorla. *Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)*. RFC 5763. RFC Editor, May 2010.

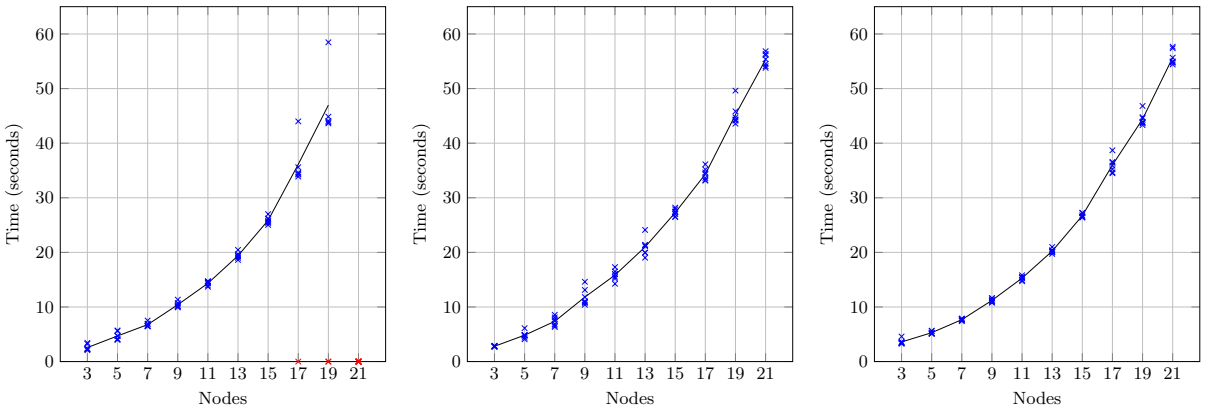
[13] Mark Handley and Van Jacobson. *SDP: Session Description Protocol*. RFC 2327. http://www.rfc-editor.org/rfc/rfc2327.txt. RFC Editor, Apr. 1998. URL: http://www.rfc-editor.org/rfc/rfc2327.txt.

[15] Emil Ivov, Eric Rescorla, and Justin Uberti. *Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol*. Internet-Draft draft-ietf-mmusic-trickle-ice-02. http://www.ietf.org/internet-drafts/draft-ietf-mmusic-trickle-ice-02.txt. IETF Secretariat, Jan. 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-mmusic-trickle-ice-02.txt.

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

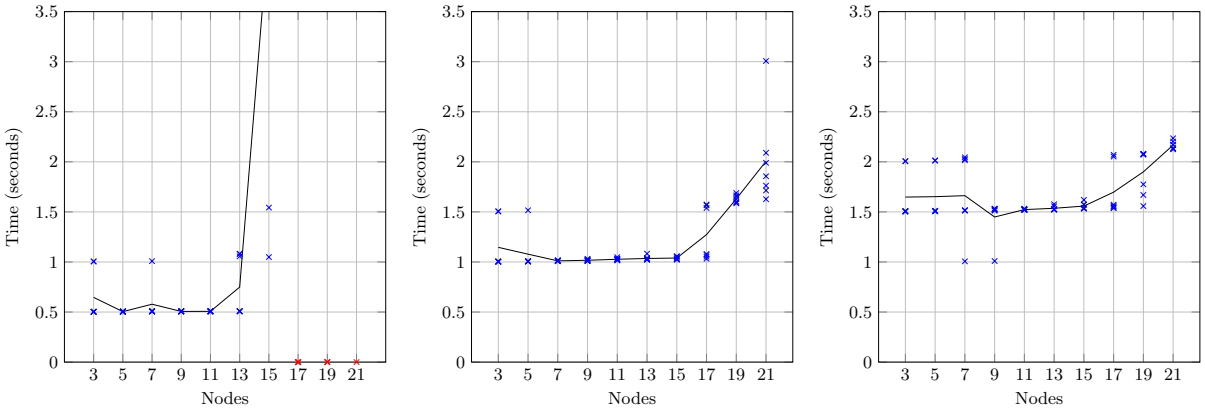
[37] J. Rosenberg and H. Schulzrinne. *An Offer/Answer Model with Session Description Protocol (SDP)*. RFC 3264. http://www.rfc-editor.org/rfc/rfc3264.txt. RFC Editor, June 2002. URL: http://www.rfc-editor.org/rfc/rfc3264.txt.

[48] Justin Uberti, Cullen Jennings, and Eric Rescorla. *Javascript Session Establishment Protocol*. Internet-Draft draft-ietf-rtcweb-jsep-10. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-jsep-10.txt. IETF Secretariat, June 2015. URL: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-jsep-10.txt.



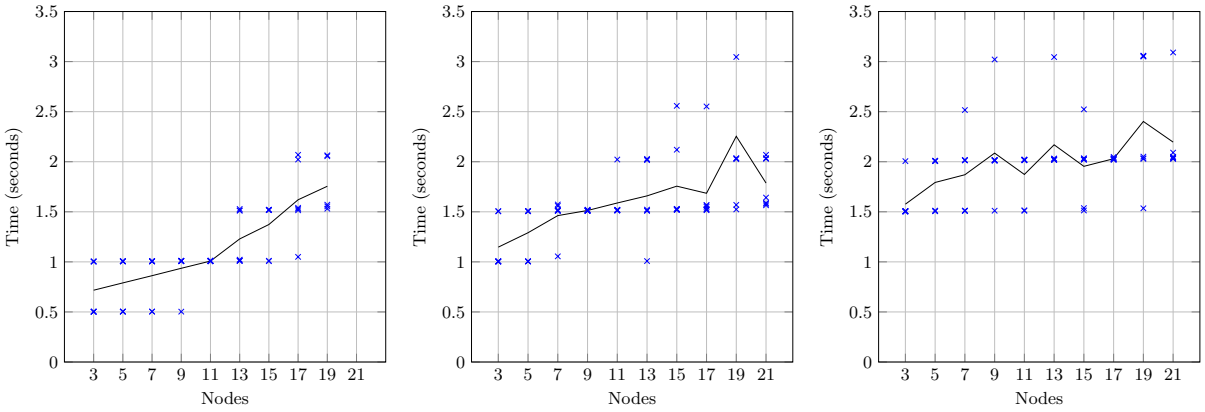
(a) 100ms election timer (b) 500ms election timer (c) 1000ms election timer

Figure 8: Initial Cluster Activation



(a) 100ms election timer (b) 500ms election timer (c) 1000ms election timer

Figure 9: Recovery from Leader Failure



(a) 100ms election timer (b) 500ms election timer (c) 1000ms election timer

Figure 10: Recovery from Half Cluster Failure