Performance evaluation of Paxos Consensus algorithm in browser using WebRTC

GIBSON CHIKAFA
chikafa@kth.se

August 30, 2021

Abstract

Achieving consensus is a fundamental problem in the space of distributed computing. On P2P networks it has led to the development of blockchain applications such as Bitcoin. While public blockchains(e.g Bitcoin and Etherium), use a different class of consensus algorithms, e.g PoW(Proof of Work) and PoS(Proof of Stake), in private blockchains where nodes are trusted traditional consensus algorithms such as Paxos and Raft can be used. Private blockchains are becoming more popular and industry use cases include asset management, capital markets and real estate. To meet the expectation of users who are used to the convenience of web applications, users should be able to connect to the blockchain by only using the browser. In this project we explore the potential of achieving consensus on WebRTC using Paxos consensus algorithm in the end-users browser. We implement the Paxos algorithm and measure its performance in terms of CPU utilization and latency. The experiment shows that CPU utilization and System Latency increases exponentially when the number of peers in the network is increased.

Code for this project is available at https://github.com/gibchikafa/paxos_webrtc

Contents

1 Introduction **3**
1.1 WebRTC . . . 3
1.2 Consesus . . . 4
1.3 Paxos Algorithm . . . 4
1.4 Research Question . . . 4
1.5 Related Work . . . 5

2 Implementation **5**
2.1 Signaling Servers . . . 5
2.2 Building a Cluster . . . 5
2.3 Paxos Algorithm . . . 5
2.4 Key Value Store . . . 5

3 Research Methodology **6**
3.1 Case Study Design . . . 6
3.2 Experimental setup . . . 7
3.3 Data Collection . . . 7

4 Results **7**

5 Conclusion and Discussion **8**

A Acronyms **9**

List of Figures

1 Establishing connection in WebRTC [1], . . . 3
2 Sequence diagram describing cluster construction . . . 6
3 System Latency as total time taken for all peers to commit their messages. . . . 7
4 CPU utilization recorded using Google Chrome Task Manager for a selected tab. . . . 7
5 Abortable Paxos Algorithm - Proposer [2] . . . 10
6 Abortable Paxos Algorithm 1 - Learner and Acceptor [2]. . . . 11

1 Introduction

In this chapter the theoretical background, research question, and related work are discussed. Section 1.1, 1.2, and 1.3 covers the theoretical background of WebRTC, Consensus and Paxos algorithm respectively. In Section 1.4 our research question is discussed. Section 1.5 covers the related work.

1.1 WebRTC

Web Real-Time Communication(WebRTC) is a set of protocols and standards that enable web applications to communicate directly without the need for an intermediate server or browser plugins [3]. The major components of the WebRTC API as mentioned in [3][4] are: (a) MediaStream: allows a web browser to access the camera and microphone, (b) RTCPeerConnection: allows browser-to-browser to communicate directly and (c) RTCDataChannel: allows browsers to send data connections and enables the exchange of arbitrary data between them. WebRTC thus allows peer-to-peer connections that can share multi-media data such as video, audio or arbitrary binary data in real-time. WebRTC has been used for example to develop video conferencing applications [5].

Estabishing Connections in WebRTC

In order o establish a connection, both peers need to provide an ICE(Internet Connectivy Establishment) configuration for the initial exchange of connectivity information [6]. This transferring of ICE candidates is known as *signaling* and the intermediary service is known as *Signaling server*. The Signaling server can be a STUN(Session Traversal Utilities for NAT) or a TURN(Traversal Using Relay NAT) server and the WebRTC API supports both STUN and TURN directly. However, one can also implement their own signaling server as we will see later in the implementation.

When a peer wants to join the network, we will call this peer a *calling* peer, it will send its ICE candidates to the the Signaling server. The Signaling server then forwards this information to the other peers already in the network; we will call these peers *receiving* peers. When the receiving peer get the ICE candidate of the calling peer it registers it and sends its own ICE candidate to the Signaling server and the Signaling server forwards this information to the calling peer. When both peers have the ICE candidate information of each other, they can directly establish peer-to-peer connection with each other without any involvement of the Signaling server. This is described in Figure 1.

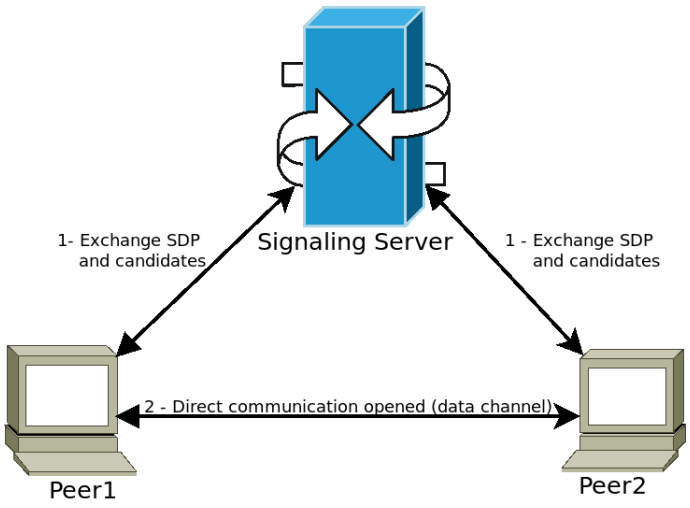

Figure 1: Establishing connection in WebRTC [1],

1.2 Consesus

Consensus is defined as the task of getting all processes in a group to agree on some specific value based on the votes of each processes [7]. Achieving consensus is a fundamental problem in the space of distributed computing. Example applications in which consensus is applied include database transactions to commit in a specific order, state machine replication, and atomic broadcasts. Real-world applications that require consensus include cloud computing, clock synchronization and load balancing. In P2P networks it has led to the development of blockchain applications such as Bitcoin. While public blockchains(e.g Bitcoin and Etherium), use a different class of consensus algorithms, e.g PoW(Proof of Work) and PoS(Proof of Stake), in private blockchains where nodes are trusted traditional consensus algorithms such as Paxos and Raft can be used. Private blockchains are becoming more popular and industry use cases include asset management, capital markets and real estate [8]. To meet the expectation of users who are used to the convenience of web applications, users should be able to connect to the blockchain by only using the browser. Nimiq[9] is a cryptocurrency that can be used on the web, but for mining tasks it is slow and they recommend using the desktop application.

1.3 Paxos Algorithm

Paxos algorithm addresses the problem of achieving a single value v given a set of processes in a partially-synchronous system model. The requirement is that all processes should "decide" on the same value, such that the following properties hold:

1. **Validity**: Only proposed values may be decided
2. **Uniform Agreement**: No two processes decide different values.
3. **Integrity**: Each process can decide a value at most once.
4. **Termination**: Every correct process eventually decides a value.

There are mainly two versions of the Paxos algorithm: (1) Single value Paxos (2) Sequence Paxos. The main difference is that in single value Paxos only a single value is chosen from values proposed by one or more process while in Sequence Paxos a sequence of values or commands is chosen and those process that contain the same sequence are said to be consistent. Both version can be leader based or without a leader. In this project we will implement the leaderless Paxos algorithm. In the leaderless Paxos each process plays one or more, quite often all, of the following roles:

1. **Proposer**: Wants a particular proposed value to be decided
2. **Acceptor**: Acknowledges acceptance of proposed values.
3. **Learner**: Decides based on acceptance of values.

For more details how the algorithm works see [2].

1.4 Research Question

To leverage the convenience of web applications, users should be able to connect to a blockchain using their browser. Connecting to blockchain on P2P network using WebRTC and achieving consensus means that the consensus algorithm needs to be run in the browser. Our research question is: How can the Paxos algorithm perform running it in the browser in-terms of latency and CPU utilization?

1.5 Related Work

Consensus in the browser using WebRTC using the Raft algorithm [10] has been studied in [11]. In this study the the latency of system (i.e., the number of committed messages) over the cluster size was measured for a span of 10 seconds. The results show that performance of the system degrades rapidly as the cluster size increases. The degraded performance results from current browsers' limitations around maintaining large numbers of RTC connections.

2 Implementation

This whole application was developed in TypeScript and NodeJS. To deal with events and asynchronous message passing we use a reactive RxJS [12] which is Javascript library for reactive programming using Observables. Implementation of important components are described in the subsections to follow.

2.1 Signaling Servers

We implement our own signaling server using NodeJS backend service which handles clients over websockets. When communicating with our signaling server, clients must either provide a target peer uuid for their message to be forwarded or specify their intent to broadcast to all registered clients. The signaling server is also responsible for generating client ids, keeping track of active clients, and handling client disconnects.

2.2 Building a Cluster

We designed a simple protocol that allows peers in the network to discover each other without predefined information. We defined five messages: *join_cluster*, *sending_signal*, *peer_joining*, *returning_signal*, and *returned_signal*. When the client wants to join the cluster, it first sends a *join_message* to the signaling server. The signaling server will first reply to the client, the message identifier is *my_peer_id*, with a unique id which becomes the client id. The id is basically the *socket id*. The signaling server will then send the client, the message identifier is *join_message*, with the socket ids of the other clients already in the cluster. When the client gets the list of other client ids, it then sends each to each client in the list its signaling information via the signaling server. The message identifier for this message is *sending_signal*. The server will forward this information to other client with message *peer_joining*. When the other client gets the *peer_joining* message replies back to the server with its signaling information. The message identifier for this response is *returning_signal*. The server will forward this information to the client that initiated the connection. At this point, every node in the cluster has the necessary information to establish peer to peer connections with each other. When the total number of clients in the cluster has reached the threshold required the Paxos algorithm can begin.

2.3 Paxos Algorithm

The Single value Paxos algorithm was implemented following algorithm 1 in [2]. Also in appendix in Figure 5 and Figure 6. For simplicity we assume Perfect Links i.e., no message loss. The Signaling server takes the role of the Failure detector. Each peer maintains a websocket connection with the Signaling server such that a disconnection is detected by the Signaling server. When a peer disconnects from the cluster, the Signaling server will broadcast a message to the rest of the peers in the cluster and is removed from the list of peers that each peer maintains. In this way an updated quorum is always maintained at each process.

2.4 Key Value Store

Each client in the cluster will generate a predefined number of messages it would like propose during the Paxos algorithm. Each client will store an in memory log of committed messages during the Paxos

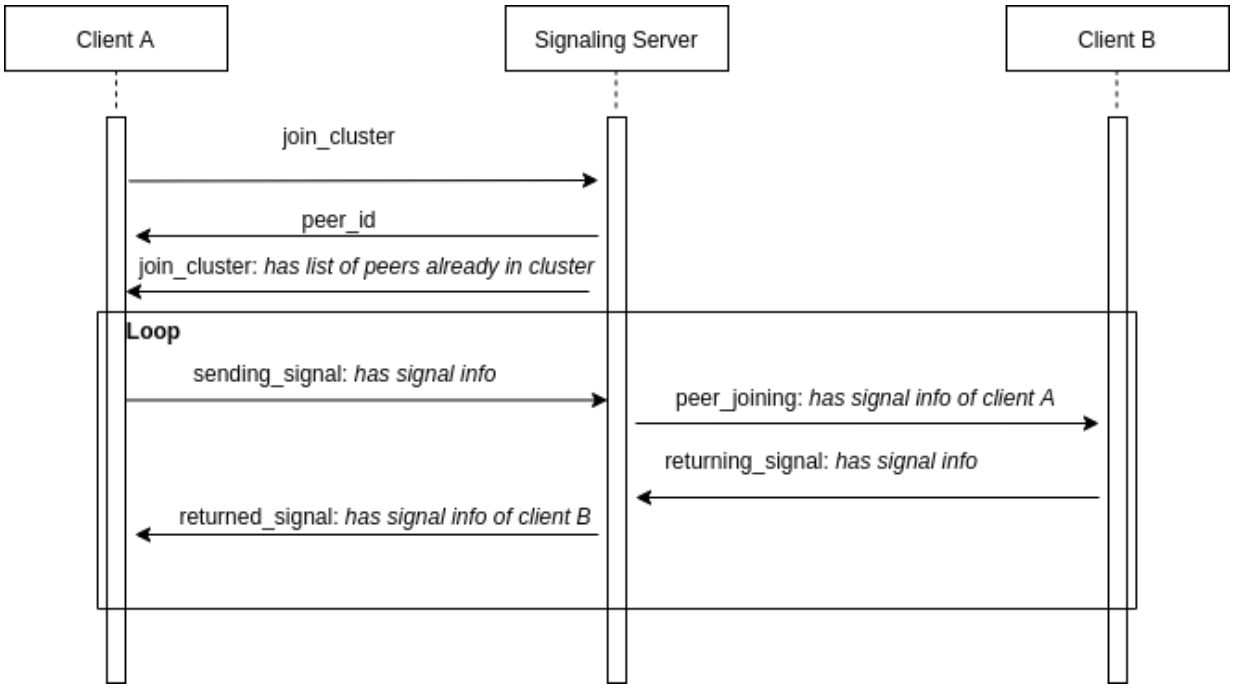

Figure 2: Sequence diagram describing cluster construction

algorithm. Each client log will contain only messages committed during the Paxos algorithm. After the Paxos algorithm terminates each client will send its log to the signaling server for verification. To verify the correctness, we compare the logs from each client if they are the same.

3 Research Methodology

We will use a combination of fundamental research method [13] and experimental research method [13]. The fundamental research method is suited because we would like to develop a proof-of-concept system and perform tests on it to get the desired system properties i.e satisfy the safety properties of the Paxos algorithm. The experimental research method is used for investigating systems' [13] performances which is what we also desire; we would like to establish the relationship between the cluster size, latency and CPU utilization when running the Paxos algorithm.

3.1 Case Study Design

To measure the system latency, the time taken to commit all the messages from each peer versus the number of peers or cluster size will be recorded. Each peer in the network will generate a predefined number of random messages it can propose and finally commit. The start time is the time when the first value is proposed by any client. The end time is the time when the last message is committed and the algorithm terminates. This time is will be recorded by Signaling server because it has knowledge of the cluster size and finally receives the log messages from each client therefore it knows when the algorithm starts and terminates. We configure each peer to propose a message every 500ms-1000ms. Each peer will propose 100 messages in total. A total of 5 trials were conducted for this setup for 4, 8, and 16 processes.

To measure the CPU utilization we used the Google Chrome Task Manager to record the CPU utilization of a client. We assume that the CPU usage of a client will be the same since each client is executing same program. Similarly, we configure each peer to propose a message every 100ms-200ms so that a significant percentage of the CPU can be utilized and recorded. Each peer will propose 100 messages in total for 4, 8, and 16 processes.

3.2 Experimental setup

The experiments were conducted on a Lenevo Legion with 16GB RAM, 6 cores, and Intel core i7 10th gen processor. The browser used was Chrome. A peer or a process is represented by a tab in the browser. The minimum number of processes required or the cluster size for the Paxos algorithm to begin is firstly configured. When such number of tabs are opened the algorithm begins to run.

3.3 Data Collection

After conducting the Case study the output for the system latency and CPU utilization were collected. The system latency time was recorded to be the time difference when the algorithm starts(i.e., first message is proposed) to the time when the algorithm terminates(i.e., last message is committed). For CPU utilization data is collected from the Google Chrome task manager which shows the CPU utilization of a particular opened tab.

After all trials are conducted for each cluster size, the averages were computed together with the errors.

4 Results

The results for system latency is shown in Figure 3. In this experiment each client commits 100 messages. Each peer can propose a message every 500ms-1000ms. The results for CPU utilization for a single peer

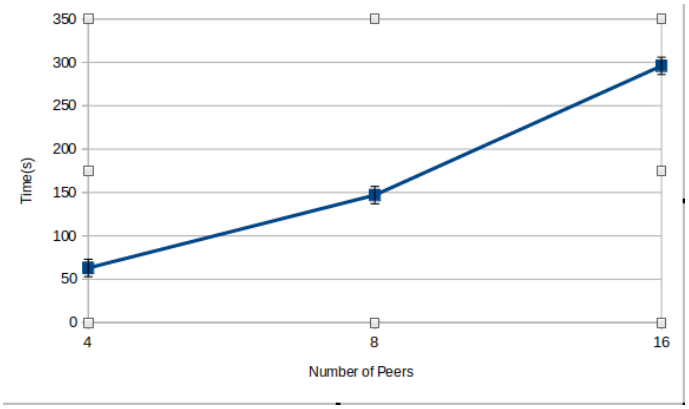

Figure 3: System Latency as total time taken for all peers to commit their messages.

was recorded using the browser Task Manager and shown Figure 4. In this experiment each client commits 100 messages and can propose a message every 100ms-200ms.

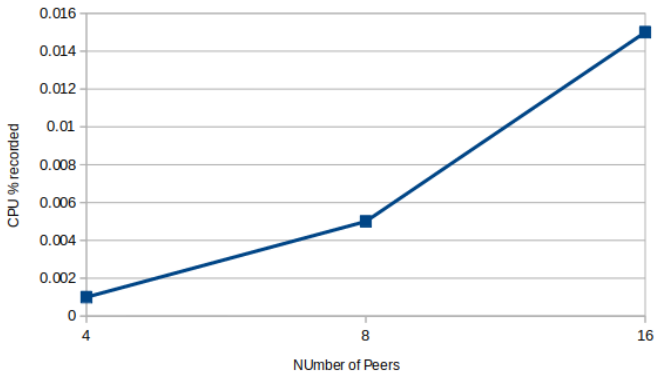

Figure 4: CPU utilization recorded using Google Chrome Task Manager for a selected tab.

5 Conclusion and Discussion

As seen from the results, the CPU utilization and System Latency increases exponentially when the number of peers in the network increases. For CPU utilization, it was further observed that if the time a single peer can proposed was made smaller the CPU utilization was high because due to large number of messages and communication processed in a short time. However, our experiment has some limitations. The experiment was almost performed manually by opening multiple tabs in the browser and therefore it was difficult to simulate a large number of peers. Above 16 tabs the browser was crashing due to limited capability to support larger number of WebRTC connections. This should have been done with some automated tool to simulate the tabs (i.e a peer in the network). Furthermore, to simulate the network latency close to a real world scenario, the experiment whould have been performed on a real network (e.g LAN) with multiple independent computers.

Future Work

Our code for the Leaderless single value Paxos can be extended to implement Leader-based single value and Sequence consensus Paxos. It would be interesting to see how these versions of the algorithm.

References

[1] *HTML5, WebSocket, WebRTC*, 2021 (accessed June 08, 2021), https://docs.godotengine.org/en/stable/tutorials/networking/webrtc.html.

[2] S. Haridi, L. Kroll, and P. Carbone, "Lecture notes on leader-based sequence paxos – an understandable sequence consensus algorithm," 2020.

[3] *WebRTC*, 2020 (accessed September 01, 2020), https://webrtc.org/.

[4] B. Sredojev, D. Samardzija, and D. Posarac, "Webrtc technology overview and signaling solution design and implementation," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2015, pp. 1006–1009.

[5] N. M. Edan, A. Al-Sherbaz, and S. Turner, "Design and evaluation of browser-to-browser video conferencing in webrtc," in *2017 Global Information Infrastructure and Networking Symposium (GIIS)*, 2017, pp. 75–78.

[6] *Getting started with peer connections*, 2020 (accessed September 08, 2020), https://webrtc.org/getting-started/peer-connections.

[7] *Consensus*, 2011 (accessed August 30, 2021), https://people.cs.rutgers.edu/~pxk/417/notes/content/consensus.html.

[8] *Blockchain Use Cases and Applications by Industry*, 2020 (accessed September 08, 2020), https://consensys.net/blockchain-use-cases/.

[9] *Nimiq: A simple, secure and censorship-resistant payment protocol, native to the web*, 2020 (accessed September 08, 2020), https://www.nimiq.com/whitepaper/.

[10] *The Raft Consensus Algorithm*, accessed August 30, 2021, https://raft.github.io/.

[11] *Consensus in the Browser using WebRTC*, 2019 (accessed September 08, 2020), http://www.scs.stanford.edu/20sp-cs244b/projects/Consensus%20in%20the%20Browser.pdf.

[12] *RxJS: Reactive Extensions Library for JavaScript*, 2021 (accessed June 08, 2021), https://rxjs.dev/.

[13] *Portal of Research Methods and Methodologies for Research Projects and Degree Projects.*, 2013, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960.

A Acronyms

WebRTC: Web Real-Time Communication

LAN: Local Area Network

JSON: Javascript Object Notation

P2P: Peer to Peer

ICE: Interactive Connectivity Establishment

Algorithm 1: Abortable Paxos – Proposer

Implements: Uniform Consensus
Requires: Perfect Link
Algorithm:

A ; /* set of acceptors */
L ; /* set of learners */
$s \leftarrow 0$; /* local sequence number */
$(n_p, v_p) \leftarrow (\perp, \perp)$; /* unique round number and value */
$promises \leftarrow \emptyset$;
$acks \leftarrow 0$;

Upon $\langle$PROPOSE $\mid v\rangle$
  $s \leftarrow s + 1$
  $n_p \leftarrow$ UNIQUE(s) ; /* use pid to make n globally unique */
  $v_p \leftarrow v$
  $promises \leftarrow \emptyset$;
  $acks \leftarrow 0$;
  foreach $a \in A$ **do**
    send $\langle$PREPARE $\mid n_p\rangle$ **to** a;

Upon $\langle$PROMISE $\mid n, n', v'\rangle$ **from** a **s.t.** $n = n_p$
  $promises \leftarrow promises \cup \{(a, n', v')\}$; /* add a for acceptor disambiguation */
  if $|promises| = \left\lceil \frac{|A|+1}{2} \right\rceil$ **then**
    $v \leftarrow$ MAXVALUE$(promises)$; /* value with the largest n */
    $v_p \leftarrow$ **if** $v \neq \perp$ **then** v **else** v_p; /* adopt v if present */
    foreach $a \in A$ **do**
      send $\langle$ACCEPT $\mid n_p, v_p\rangle$ **to** a;

Upon $\langle$ACK $\mid n\rangle$ **from** a **s.t.** $n = n_p$
  $acks \leftarrow acks + 1$;
  if $acks = \left\lceil \frac{|A|+1}{2} \right\rceil$ **then**
    foreach $l \in L$ **do**
      send $\langle$DECIDE $\mid v_p\rangle$ **to** l;

Upon $\langle$NACK $\mid n\rangle$ **from** a **s.t.** $n = n_p$
  ABORT() ; /* Goto $\langle$PROPOSE $\mid v\rangle$ and pick a new n_p immediately to avoid old messages being handled */

Figure 5: Abortable Paxos Algorithm - Proposer [2]

Algorithm 1: Abortable Paxos – Acceptor

Implements: Uniform Consensus
Requires: Perfect Link
Algorithm:

1: $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
2: $(n_a, v_a) \leftarrow (\perp, \perp)$; /* sequence number and value accepted */

3: **Upon** $\langle$PREPARE $\mid n\rangle$ **from** p
4: **if** $n_{prom} < n$ **then**
5: $n_{prom} \leftarrow n$;
6: **send** $\langle$PROMISE $\mid n, n_a, v_a\rangle$ **to** p;
else
7: **send** $\langle$NACK $\mid n\rangle$ **to** p; /* optimisation only */
8: **Upon** $\langle$ACCEPT $\mid n, v\rangle$ **from** p
9: **if** $n_{prom} \leq n$ **then**
10: $n_{prom} \leftarrow n$;
11: $(n_a, v_a) \leftarrow (n, v)$;
12: **send** $\langle$ACCEPTED $\mid n\rangle$ **to** p;
else
13: **send** $\langle$NACK $\mid n\rangle$ **to** p; /* optimisation only */

Algorithm 1: Abortable Paxos – Learner

Implements: Uniform Consensus
Requires: Perfect Link
Algorithm:

1: $v_d \leftarrow \perp$; /* decided value */

2: **Upon** $\langle$DECIDE $\mid v\rangle$
3: **if** $v_d = \perp$ **then**
4: $v_d \leftarrow v$;
5: **trigger** $\langle$DECIDE $\mid v_d\rangle$;

Figure 6: Abortable Paxos Algorithm 1 - Learner and Acceptor [2].