PaxStore

CS7610 Project Report

Girik Malik
Abhidipta Sengupta
Kunal Patil

Table of contents:

Introduction

To address the audiences with modern web and to handle large amounts of data, many applications are taking the distributed systems approach. However this comes with its own set of problems such as data availability, data accuracy and minimized cost of overheads for doing basic reads and writes. The most intuitive way to approach this is by using key-value stores, since they are simple and speedy. The path to retrieve data is a direct request to the object in memory or on disk. The relationship between data does not have to be calculated by a query language; there is no optimization performed. They can exist on distributed systems and don't need to worry about where to store indexes, how much data exists on each system or the speed of a network within a distributed system they just work.

With this project, we aim at building our own implementation of a distributed key-value store. With each distributed system, comes the problem of having consensus in the system, that is, having each server agree on a particular update; this in turn provides the consistency which is highly desired in any system. To overcome this problem, we plan to implement our key-value store over an implementation of the replicated paxos protocol. By using Paxos we can be sure that we will achieve strong consistency, fault-tolerance as well as high availability.

We are mainly focused on keeping the data consistent and available since these properties are majorly required in application scenarios like bank, military, health,etc. For these applications, any inconsistency in replicas is intolerable. In such situations, when exposed to hostile system environments, two-phase commits may not guarantee strong consistency among multiple replicas and high system availability. With three or more replicas, the Paxos family of protocols is considered to be the only solution to guarantee the strong replication consistency. High availability is also a very desirable property for systems that provide 24x7 reliable services.

Base layer - Paxos

Paxos [Lamport '98, Lamport '01] is a (set of) simple protocol(s) for a group of machines in a distributed system to agree on a value proposed by a member of the group. If it terminates, the protocol reaches consensus, despite the network being unreliable and multiple machines simultaneously trying to propose different values. The basic idea is that each proposal has a unique number. Higher numbered proposals override lower-numbered ones. However, a "proposer" machine must notify the group of its proposal number before proposing a particular value. If, after hearing from a majority of the group, the proposer learns one or more values from previous proposals, it must reuse the same value as the highest-numbered previous proposal. Otherwise, the proposer can select any value to propose.

Paxos involves multiple trade-offs between number of machines in the group, number of messages delayed before reaching the agreed value, states of individual machines, number of messages sent, types of failures, etc.

Paxos' primary advantage is the safety (consistency) guarantee, and the design decisions that could prevent it from making progress are difficult to provoke. In general, deterministic fault-tolerant consensus protocols do not guarantee progress in asynchronous environments. It also has mechanism of permanently dropping a failed replica or adding new ones.

Paxos is still widely used in database and file replication, where consistency and durability guarantees are often needed, even in case of unresponsive behaviours; more than speed, and real-time behaviours. Google's MegaStore is one good example where Paxos is known to be performing at scale, apart from other financial applications.

Problem Statement

Design a consistent, highly available distributed key value storage system that can run on lots of general devices and solve the three problems in distributed systems: Consistency, Availability and Fault Tolerance.
Consistency : For any distributed storage system, data consistency is one of the major factors to take care of. All updates should be present in all the replicas/nodes in the system and a simple two- phase commit protocol is not enough to provide this consistency.
Availability : With distributed systems, network partitions are unavoidable, and as we know from the CAP theorem, a system where partition is allowed, we cannot maintain both C and A.
Fault Tolerance : Again, with distributed systems, server crashes are quite common and recovering those crashed nodes with previous context is another challenging task.

System Architecture

Related Work

There are a number of distributed data stores which operates along the same line, and provides the three aspects of CAP property in various ways. For example, Cassandra provides high availability and partition tolerance by using eventual consistency, or Dynamo uses the Quorum mechanism to manage replicas, which is a decentralized system.

Approach

Our approach has two major components - (i) Paxos algorithm and (ii) A Key Value Store

1. Paxos

Leader Election:

- Every server waits for a certain time period before participating in an election phase.This time-period is maintained by a progress timer which we have implemented with threads in Python
- Every server gets to be the leader in a round-robin fashion and for a certain round the leader is selected by the equation, *server_id == i mod N*, where server_id is an id assigned to a particular server, i is the current view id and N is the total number of servers in the network
- New view to be installed in communicated with *VC_Change* messages, sent and received by every server and lagged behind servers are brought up to speed by *VC_Proof* messages , sent periodically by every server to every other server.
- We move on to the next phase when the majority of the servers agree on the new view

Prepare Phase:

- This phase ensures safety across view changes.

- In this phase the leader sends a *Prepare* message to all other servers with its local ARU value, and every other server sends a *Prepare_OK* message back to the leader with new *Proposal/Globally ordered Updates* messages they have received and in this the leader learns about any new proposals/updates

Proposal Phase:

- In this phase, the updates are globally ordered by the leader.
- The leader sends *Propose* messages to all other servers with last known accepted proposal or first update in update queue and all other servers responds back with an *Accept* message.
- The leader, on receiving a majority of the accept messages for a particular view ,the leader orders the update and all other servers can now execute the update
- The proposal messages contains the client update that needs to be globally ordered. Each server stores the proposal message along with the client update in its global history data structure at a particular sequence number. This sequence number is the update's globally ordered sequence number.

Accept Phase:

- On receiving a proposal for a particular sequence number, the server prepares an Accept message and sends it to all the servers in the system.
- This does not include the client update since all servers should already have the client update with the proposal that they received.
- This message contains the sequence number, the server id and the view for which the client update is accepted.
- Each server checks for a majority of accept messages before moving on to the next step.

Client Update Execution:

- Once a majority has been received, the client update will be executed on each server. This means, each server will write the key-value pair to it's local database (key-value store).
- Since Paxos is run only for any "POST" call (write/update data), client read instructions are pretty fast and have very less latency given that the server the client is connected to has received the data yet.

- On the client side, the client sends a *get <key>* in order to retrieve the value for a particular key. The client can also send a *iget <location of image>* to retrieve the metadata for an image
- Similarly, to write data we can use ***set*** *<key> <value>* to store a string key value pair, and ***iset*** *<location of image> <related metadata>* to store an image hash and its metadata.
- Once the client update has been executed, the progress timer is restarted so the view stays the same and the current leader can continue being the leader.

Reconciliation:

- Every server sends its local aru value periodically to all the other servers. The local aru value reflects the sequence number of the last update that the server executed.
- If a server receives an aru value that is greater than its own local aru value, then it will request for the missed updates from a random server. This is usually just replying to the first aru message received.
- On receiving this request, the other server will send all the updates that have been missed by the requesting server, and the requesting server can immediately execute these updates since they have been already globally ordered.

Recovery:

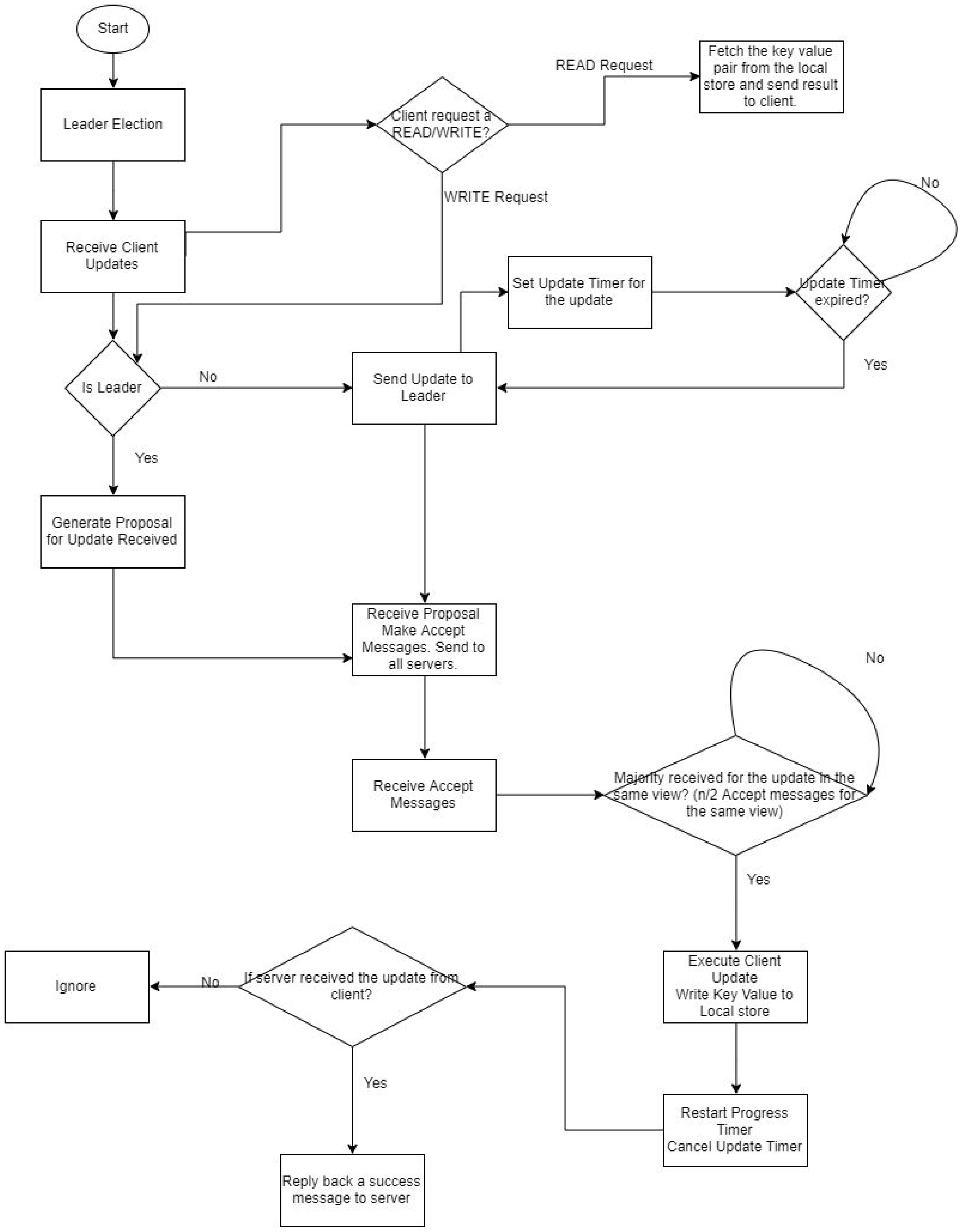
- This aspect of the algorithm deals with the scenario where a server crashes and after revival, joins the algorithm at the current ongoing stage
- We have maintained a static data structure for every server , which stores all the meta-data about the server and the current states about the algorithm and this data is written in local file at specific situations. Thus, whenever a server revives, it can simply read from this local file and revive its state just before crashing.

2. Key Value Store

Key-value store builds on top of the consensus provided by Paxos at the bottom layer. We implemented a hashing algorithm for data storage, similar to Python's dictionaries. We did this in part to keep our implementation as close and generic to the native dictionary as possible. We provide functions to get and set

key-value pairs, among other functions to modify the underlying data structure implementation.
Further, we write the data to disk after every set, by default, but that can be changed to write periodically if the hardware supporting the data structure provides reliability guarantees. These disk write are to be used in case of recovery. The implementation is based on a mix of both protocol buffers and a database-style key-value store. Protocol buffers (v2.0) are used to generate the basic structure for holding the key-value data, topped with database-style methods and subroutines for modifications to the data structure.
As an extension, we also store data using image as key, wherein we hash the image and store given metadata as value. It is possible to fetch the metadata for corresponding image by providing the image location. We use a hash function called “Average Hash”, that works by first converting the given image to grayscale, and then scaling it down to a chosen size. It then calculates the average of all gray values in the image, and starts comparing individual pixels from left-to-right, adding 1 to the hash if the gray of individual pixel is greater than the average, and 0 otherwise. Just by way of functioning, average hash also provides storage improvements, by hashing the exact same color and grayscale image to the same key. As a word of caution, one might want to use a custom image hash function in the unlikely case of color and grayscale images representing different things.
The feature of hashing by images as key, might be useful for applications in machine learning, particularly in computer vision, or for a simple implementation of lookup by image. In furtherance to this, if one wants to query based on patterns in multiple images, one can simply work with the hash of multiple images at once to decipher certain structures, providing significant computational efficiency.

State Diagram

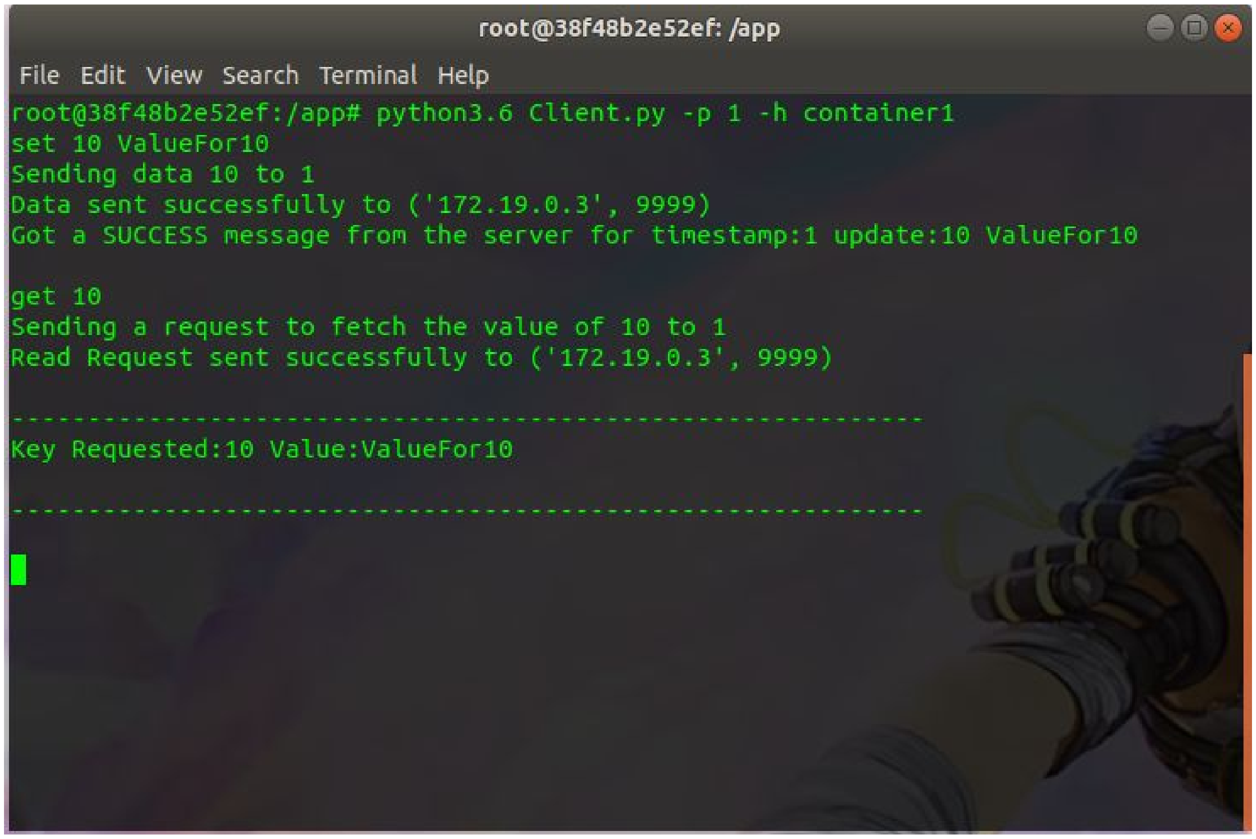

Start
Leader Election
Receive Client Updates
Client request a READ/WRITE?
READ Request
Fetch the key value pair from the local store and send result to client.
WRITE Request
No
Set Update Timer for the update
Update Timer expired?
Yes
Is Leader
No
Send Update to Leader
Yes
Generate Proposal for Update Received
Receive Proposal Make Accept Messages. Send to all servers.
Receive Accept Messages
No
Majority received for the update in the same view? (n/2 Accept messages for the same view)
Yes
Execute Client Update Write Key Value to Local store
Restart Progress Timer Cancel Update Timer
If server received the update from client?
No
Ignore
Yes
Reply back a success message to server

Design Decisions

- We decided to use Update Timers for each client update. This way, if a client update has not been ordered globally, the server responsible for it resends the client update to the leader
- On the client side, for storing images, the client code fetches the image from the given location and performs an average hash over it, and then sends this hash to the servers for writing it with its metadata.
- Progress timers are used for detecting leader failures. These timers are reset every time a leader makes some progress (or schedules a client update)
- We decided to use a class to maintain all the important data for a server. This is called ProcessVariables. For recovery, we write the instance of this class to a file and load this instance while recovering.

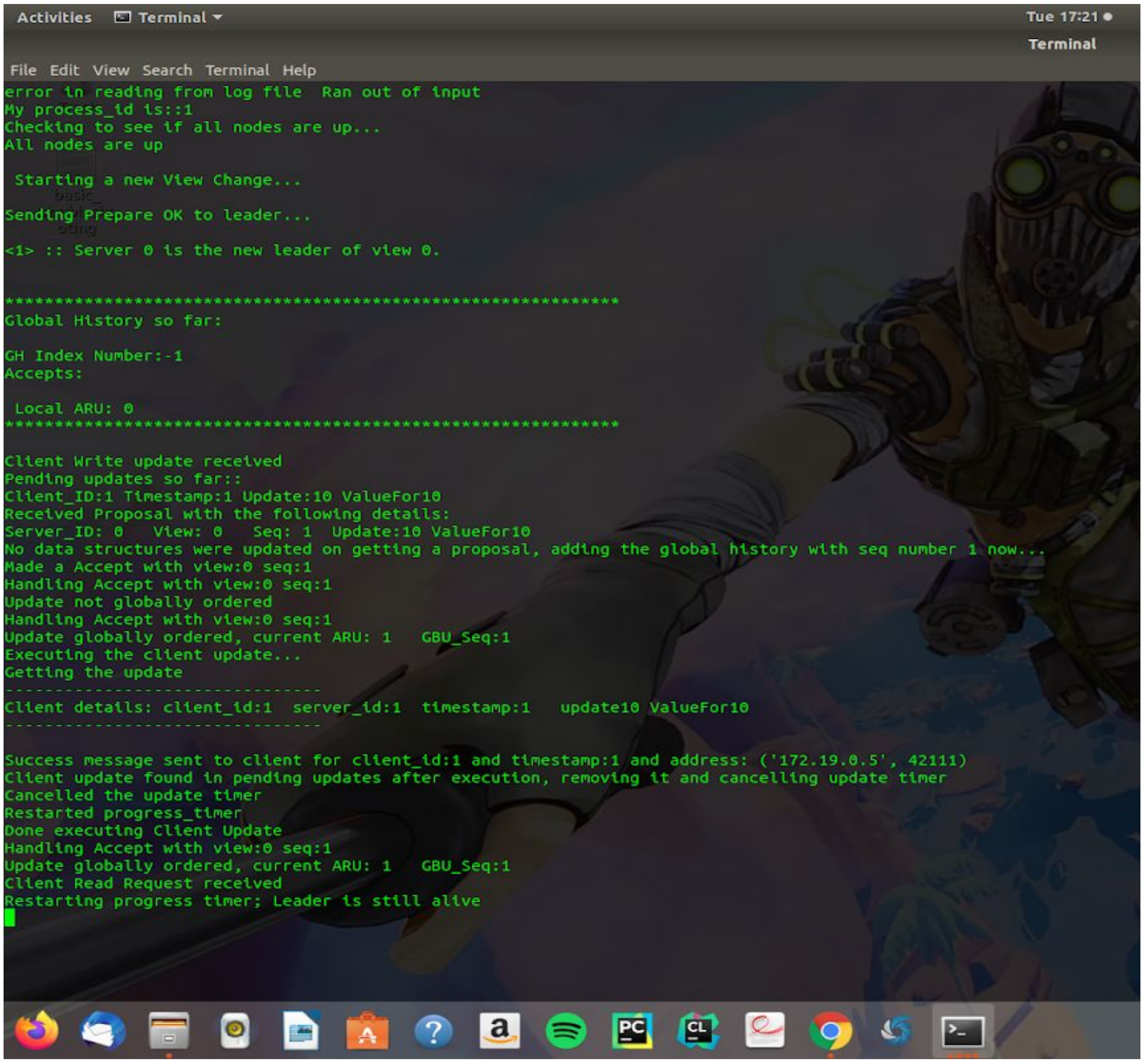
Output:

Client calling a *set* and *get* call to server (container1) :



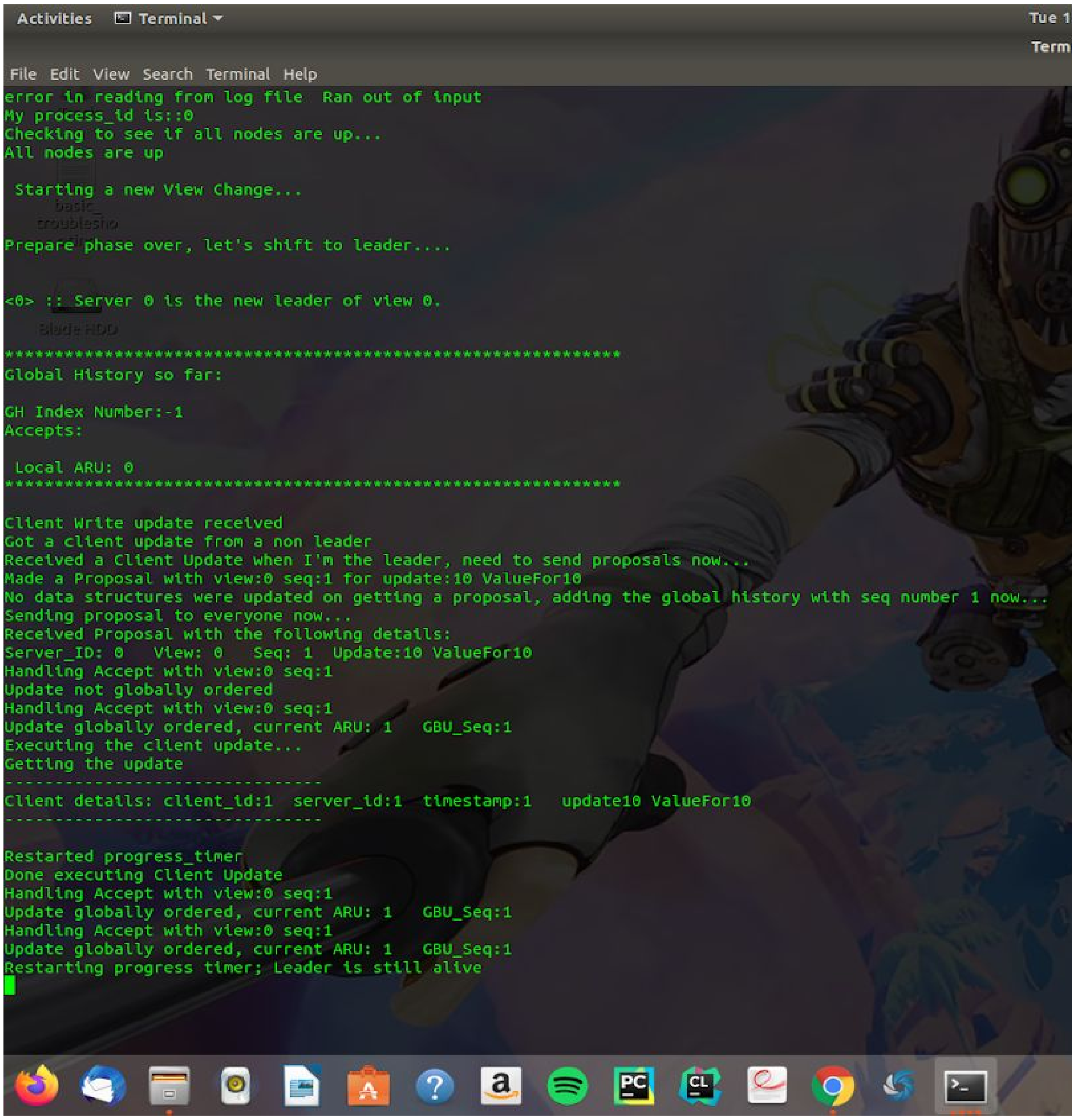
```
root@38f48b2e52ef:/app# python3.6 Client.py -p 1 -h container1
set 10 ValueFor10
Sending data 10 to 1
Data sent successfully to ('172.19.0.3', 9999)
Got a SUCCESS message from the server for timestamp:1 update:10 ValueFor10

get 10
Sending a request to fetch the value of 10 to 1
Read Request sent successfully to ('172.19.0.3', 9999)

------------------------------------------------------------
Key Requested:10 Value:ValueFor10

------------------------------------------------------------
```

Server 1 (Container1) receiving the client update, sending it to leader and then receiving the proposal, followed by sending/receiving accept messages and executing the client update:



```
error in reading from log file  Ran out of input
My process_id is::1
Checking to see if all nodes are up...
All nodes are up

 Starting a new View Change...

Sending Prepare OK to leader...

<1> :: Server 0 is the new leader of view 0.


**************************************************************
Global History so far:

GH Index Number:-1
Accepts:

 Local ARU: 0
**************************************************************

Client Write update received
Pending updates so far::
Client_ID:1 Timestamp:1 Update:10 ValueFor10
Received Proposal with the following details:
Server_ID: 0   View: 0   Seq: 1  Update:10 ValueFor10
No data structures were updated on getting a proposal, adding the global history with seq number 1 now...
Made a Accept with view:0 seq:1
Handling Accept with view:0 seq:1
Update not globally ordered
Handling Accept with view:0 seq:1
Update globally ordered, current ARU: 1   GBU_Seq:1
Executing the client update...
Getting the update
--------------------------------
Client details: client_id:1  server_id:1  timestamp:1   update10 ValueFor10
--------------------------------

Success message sent to client for client_id:1 and timestamp:1 and address: ('172.19.0.5', 42111)
Client update found in pending updates after execution, removing it and cancelling update timer
Cancelled the update timer
Restarted progress_timer
Done executing Client Update
Handling Accept with view:0 seq:1
Update globally ordered, current ARU: 1   GBU_Seq:1
Client Read Request received
Restarting progress timer; Leader is still alive
```

Server0 (Container0) - Leader of the current view, receiving the update and sending proposals:



```
error in reading from log file  Ran out of input
My process_id is::0
Checking to see if all nodes are up...
All nodes are up

 Starting a new View Change...

Prepare phase over, let's shift to leader....


<0> :: Server 0 is the new leader of view 0.

***************************************************************
Global History so far:

GH Index Number:-1
Accepts:

 Local ARU: 0
***************************************************************

Client Write update received
Got a client update from a non leader
Received a Client Update when I'm the leader, need to send proposals now...
Made a Proposal with view:0 seq:1 for update:10 ValueFor10
No data structures were updated on getting a proposal, adding the global history with seq number 1 now...
Sending proposal to everyone now...
Received Proposal with the following details:
Server_ID: 0   View: 0   Seq: 1  Update:10 ValueFor10
Handling Accept with view:0 seq:1
Update not globally ordered
Handling Accept with view:0 seq:1
Update globally ordered, current ARU: 1   GBU_Seq:1
Executing the client update...
Getting the update
--------------------------------
Client details: client_id:1  server_id:1  timestamp:1   update10 ValueFor10
--------------------------------

Restarted progress_timer
Done executing Client Update
Handling Accept with view:0 seq:1
Update globally ordered, current ARU: 1   GBU_Seq:1
Handling Accept with view:0 seq:1
Update globally ordered, current ARU: 1   GBU_Seq:1
Restarting progress timer; Leader is still alive
```

References

1) Paxos For System Builders, Jonathan Kirsch, Yair Amir. Technical Report CNDS-2008-2.