6.824 Project Report

Collaborative Drawing Pad

Group Member: Shiyang Liu, Yihua Li, Yixin Li

1. Summary

We built a web-based app tool for collaborative drawing online in real time. It enables multiple users to simultaneously draw in a common canvas space with low latency. The system also deals with failures of clients and servers and periodically saves its state for persistent storage.

2. System Design

2.1 System Overview

The system consists of a fixed number of servers and a variable number of clients. On the client side, the front end displays a canvas that allows users to draw strokes on it. And subsequently, the requests are sent from this single client first to its closest server, and if the call fails, the client retries with other servers until success.

Each distributed server keeps a history of strokes applied to the canvas. Upon receiving a draw request from a client, the server uses Paxos(Mencius) to reach agreement for that instance, fills in the empty slots in the log before this decided instance and finally replies to the client. The server also periodically stores its state to the disk for persistent storage.

The client periodically pings the server to find the highest instance number decided, gets missing instances, if any, and applies them to its own canvas.

2.2 System goal

First, we aim to achieve correctness and define correctness as: (i) a user's stroke won't be reflected on his/her canvas until a majority of servers have agreed on that instance; (ii) All other clients will eventually see that change; (iii) For a single client, a stroke will only be reflected on his/her canvas after a previous stroke drawn by himself/herself is shown. (iv) All clients will observe the exact same ordering of all the strokes.

Correctness is guaranteed since (i) Paxos(Mencius) eventually reaches agreement on an instance as long as a majority of servers are alive; (ii) Paxos(Mencius) guarantees that only one instance will be chosen for one slot in the log and all servers will agree on that slot. (iii) Server executes slots successively in the oder of increasing instance number. (iv) A client only sends the next draw request after the server sends back ok message for the previous stroke he/she drew.

The second goal is to achieve low latency. Servers need to handle multiple outstanding requests from all clients as multiple users should be able to update in parallel. In addition, when a client joins in the canvas, the system needs to quickly show him/she the strokes already committed.

The system also needs to store its state to the disk so that what the client has drawn is saved to persistent storage and could be retrieved if servers crash.

Finally, the system needs to deal with the failure of non-majority of servers. We assume that each machine may crash, lose its disk content or fail to respond to messages from others indefinitely, but it will always respond in a way that conforms to the protocol. In addition, packets can take arbitrarily long to be delivered through the network, can be duplicated or lost, but they are not corrupted.

3. Implementation Details

3.1 Front end and Client

The client is written in go with javascript front-end. Javascript communicates with go via *Post* request. We modified web drawing applet sketch.js (source code: http://intridea.github.io/sketch.js/) for the user interface. The user can choose the color and the size of the marker for drawing. When a user draws on the canvas, the start point, end point, color and size of the stroke is captured, sent to client via *Post* request and then sent to server via *Put* RPC.

The client periodically pings the server with *Update* RPC to find latest instances decided since last ping. Sketch.js receives these updates and apply them to the canvas with *Draw* method. The stroke a user draws will not appear on the canvas until the client gets the committed instances by all clients that he missed from the server. This design ensures that all strokes appear on the canvas in the right order. Since each client get updates from the server every at 0.2s, the user can hardly notice the delay. Finally, the client has the option of batching several strokes into one put request in order to increase throughput.

3.2 Server

One main function of the servers is to reach consensus on the ordering of strokes from the clients. A stroke has a start point, an end point, color and size. Each stroke is contained by an instance, which has a globally unique instance number indicating its position in the log of each server. Later instances in time have higher instance numbers.

To speed up the agreement process, we tried different variants of Paxos protocol. We first implemented Epaxos. An instance/stroke depends on the same client's previous stroke and all strokes that intersect with it. Servers then need to transfer their 2D log to the clients, who execute decided instance upon receipt to minimize latency. However, partly due to the fact that all strokes's dependency list are non-empty, Epaxos does not offer much speed up since the fast path is not often utilized and additional time is spent on computing dependency list. As a result, we next implemented Mencius and integrated it into the project. And please consult Section 4 for the performance of Paxos and Mencius.

The server handles two types of RPC calls from the client: *GetUpdate* and *Put*. For *GetUpdate* request, the server checks if the client has missed any instances by comparing client's maximum executed instance number to its own. If the client has not yet executed some instances that the server has executed, the server returns the missing instances in the order of increasing instance number to the client. After receiving a *Put* request, the server keeps proposing for it until it is decided among Paxos peers. Then the server executes in increasing order all previous unexecuted instances in the log. (This is the similar to lab 3 and 4) Finally, like how it replies to *GetUpdate* request, the server returns to the client a list of instances that it misses.

Each server has a go routine that periodically checks if an instance with instance number equal to its paxos.Max() has been decided and inserts a Noop operation if not. Then the server keeps executing in order all previous unexecuted instances, i.e. fill in the holes in the log.

To achieve persistent storage, the system stores both the paxos log/state and server's relevant states such as the map storing most recent client requests as files into disk through *WriteToDisk*. For performance consideration, the system could takes a snapshot only after executing a fixed number of operations instead of after each operation. This interval for taking snapshot could be set by the system administrator. We set it to be small (50 operations), so that only a small number of some recent stroke are lost and users could easily recover the lost strokes. When a crashed server restarts, it reads the files from disk and starts from the snapshot taken before the failure. If a server crashes and loses its disk content, it will be brought up to date (learn and execute previous agreed instances) when a client issues some request to it.

(Please see https://github.com/lsyang/drawingpad for the code. The Readme.txt provide instructions to run servers and clients)

4. Benchmark

1. Latency and Throughput

The throughput of Mencius is higher than that of Paxos (from lab3) for all scenarios. This is because if servers have balanced load, most of the instances only need one-round trip instead of two-round trip messages to be agreed upon.

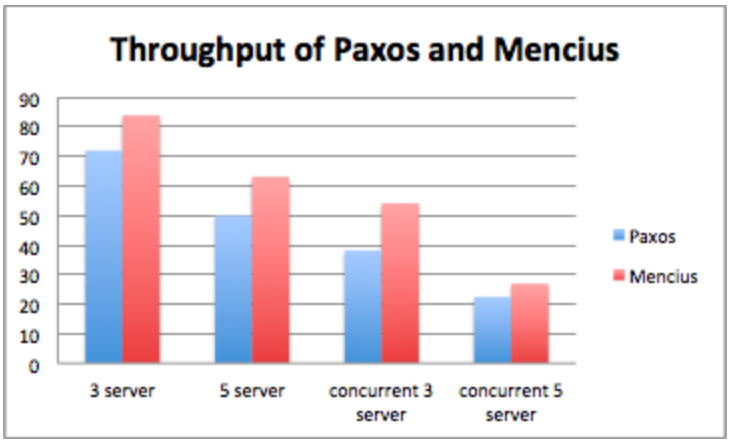

Figure 1. Throughput of Paxos and Mencius as measured in op/sec.

GetUpdate has similar latency as *Put* operation.

Latency is slightly higher when persistence storage is enabled. When 300 *Put* ops are executed concurrently with 3 servers, Paxos has latency of 0.015s/op with no storage and 0.028s/op on average with storage. This time difference is mainly due to the time needed to write operations to disk at every 50 operations interval.

2. Recovery Time

The main bottleneck in performance is recovery from disk crash. In this case, the crashed server has to obtain each of the previous paxos instances from other servers, write them to disk and display them on canvas. The whole process of recovery takes about 5 seconds.

5. Testing

In addition to the tests in lab 3b that tests for different scenarios such as unreliable network in which RPC requests and replies are lost, partitions of the servers and concurrent client request, we also tested the system's correctness in the event of server crash and the loss of its disk content under both normal and unreliable conditions. These test cases are:

1. A minority of servers crash for some period of time and then recovers.
2. All servers crash and recover later on.
3. A minority of servers crash and lose their disk content.

Note that since the server does not save until a fixed number of new instances have been decided or executed, a small number of most recent strokes are lost. And we do not check for those strokes.

We also manually tested above scenarios by drawing on the canvas, applying the failures (manually shutting down the server and/or deleting the disk content) and checking if the canvas works as expected.