The different steps of the last two month was separated into 4 main steps that will be described hereafter.

I. Build CRDT in IPFS using C++

In a first step, the goal was to develop a code that can save CRDT in IPFS following the Merkle-Dag CRDT algortihm described in it's paper. To build this CRDT there was multiple challenges. The first was to find out how to use IPFS in my code. I first discovered the ipfs implementation usable in shell. Easy to use with simple instructions. So i decided to try to use it in C++ by using system calls. This worked pretty well but show some issue of environment in the sytem calls. I solved it using tmux, a terminal manager that helped me managing virtual shell so input, output and multiple commands was easier to manage.

After this issue, the building of CRDTs in IPFS followed this idea :

- Having a CRDT type that directly represent the data but is never directly used.
- Having a CRDTDag that represent the CRDT with adding order of modifications
- Specify modification though the Payload infrastructure.
- At each new payload created, create a New Node in the DAG
- write the New Node in a File, add it to IPFS and send the CID to others via pubsub. When you receive a CID, you get the file though IPFS and then merge it with your dag using the algorithm of the MerkleDag Paper.
- when ever someone want to get the value of the data, we go though all nodes recursively so CRDT can be rebuild respecting the same order even with concurency because every node will eventually have the same DAG.

When this system is built there is one question that comes with remote connections. What happens when someone connect after some updates have already been made ?

I first used a simple solution that says that when a node connect, he send a specific phrase in the pubsub topic. When other nodes see this specific phrase, they do send their nodes so the incoming node can know it. An issue is that obviously it must flood the network for nothing.

Another solution I've implemented after but not tested is to retrieve old data using direct dependency recursively. So nodes can eventually retrieve the full state without requiring anymore messages. But it does work only if message are kept beeing sent.

II. Testing of the CRDT using C++

After implementing this system, some test was necessary. A trouble with test was that ipfs daemon was general in a session, so it was hard to generate multiple nodes on the same computer. It should have been possible by using VM's but i rather fastly used g5k to solve this issue because it manage virtualization itself.

I firstly managed multiple nodes with having multiple binaries and forcing values of topic and giving as input the ID of another peer so they connect together. Then I figured that I didn't know exactly what to measure, so i did measured the number of message sent using pubsub, as it was easy to measure.

A good point with these experience is that it showed that at least using a small number of node (3), it does converge fastly. After these experience I tryed to figure out how to manage many nodes easier by reading hadoop deployment code but it has been interupted by the Hive's mission, which lead to another work.

III. GO usage motivation

There two main issue with this implementation : - Kubo(IPFS base implementation) is designed in GO and as I use it though shell, it is hard to follow message send and received. - IPFS does implement a PUBSUB but it is still at an experimental state and haven't been tested, also it haven't been maintained since few years.

These two issue can be fixed by one common point, using the GO language, so i can use ipfs as a library and use libp2p to exchange messages.

IV. Build CRDT in IPFS using GO

As go implementation seems important, I did spent some time on learning go and a big difficulty here was also to understand how to use the IPFS library. As the documentation was not up to date and exemple wasn't always working.

Then Learning LibP2P was also a big mess as I did managed to create nodes but the way that nodes connects together was hard to manage. I did lost a lot of time trying to make work the IPFS as a lip2P node because it seemed possible. I didn't managed to do it and finally I Have used the code of Ludovic To manage node connection. And it still use libp2p pubsub

Once the issue of connection between peers was solved, It was possible to build the code as i did in C++ but in Golang. A problem was adapting what I did using the inheritance of C++ without using any inheritance as it doesn't exist in Go. I did had some basic issue that took me some time to solve, but it was classic issues.