2st Assignment – Map-Reduce w/ Fault Tolerance

Subject: Computação Distríbuida

Docentes:

- **Diogo** Gomes
- **Mário** Antunes

Students:

- **Diogo** Silva, nMec *89348*
- **Vasco** Ramos, nMec *88931*

Coordinator Asyncio

How it works [Creation]:

1. Start **Asyncio loop**
2. Create a **server** object using the Coordinator class in the loop we created
3. Make **server** in the **loop** keep looping forever (or until it crashes)

How it works [EchoProtocol Class]:

- Has a Coordinator object as a class field
- When a connection gets made we register it (*connection_mad*e() function)
- When data gets received we call the Coordinator's receive function
- When an *EOF* gets received (i.e a connection breaks) we call the Coordinator's *redistributeWork()* function

Notes:

- We used (as you could see from the methodology) ***Callback*** based Asyncio
- Only the Coordinator uses Asyncio because he's the only entity that acts as a "server", as in, every other entity connects to it
- Using Asyncio allowed us to easily and effectively implement a way for the Coordinator to know who sent it what (hence eliminating the need to implement *Selectors*)

Message Partitioning

How it works:

1. Decode the received data (using *UTF-8*)
2. Check if the control character (‘\x04’) is **NOT** present in the data received
   - 2.1 if True: Append the received data to the *self.msgBuffer* field
   - 2.2 else:
     - 2.2.1 Split the received data by the control character (‘\x04’)
     - 2.2.2 Save the result of appending our current *self.msgBuffer* to the value of the split at index 0
     - 2.2.3 Call our *handle()* function passing it the result of 2.2.2
     - 2.2.4 Store in *self.msgBuffer* the value of the split at index 1

Notas:

- This method is called everytime we receive new data
- All classes have a *self.msgBuffer* field
- This was necessary to do because *TCP* sends messages that are too large in chunks of smaller messages, so we required a way to rebuild the message on the receiver and know where one message ended and one began
- The way we solved this issue was by appending a special character ,‘\x04’, at the end of every message, thus making a sort of “flag” to know where one message ended
- All of 2.2’s proceedings are done so that, in the eventuality that we receive 2 messages at the same time (which may happen in the Coordinator/Backup when, f.ex, another message gets sent while we’re still rebuilding our current one)
- The Backup, Coordinator & Worker differ slightly in the implementation of 2.2 but the idea/methodology is the same

Backup Sync & Setup

How it works [Creation of Backup]:

1. Try to start a **Coordinator** at the default address (*127.0.0.1* , *8765*)
    1.1 if Success: Then we've created a **Coordinator** and not a **Backup** [Terminate]
2. If 1. fails (using a Try...Except)
    2.1 Increment default port by *1*
    2.2 Try to create a **Backup** instance at that port
        2.2.1 If we fail, go back to 2.1
3. Start the created backup's instance (by calling *backup.start_backup()*)
4. Backup sends the coordinator a *reg_backup* message
5. Coordinator registers the backup's connection to it in the variable *self.backupConn*
6. The coordinator sends all **Worker**s the backup's address (so that they may connect to it if they wish to do so) [The backup's port for connecting to other entities that aren't the backup is included in the *reg_backup* message]

How it works [Sync of Backup]:

1. After the coordinator does operations on it's main elements (*self.maps* & *self.datastoreIndex*) it sends an updated version of them to it's backup (if it has one) in an update message
2. Backup changes it's internal *self.maps* and *self.datastoreIndex* to the ones included in the update message

Notas:

- We allow only for the existence of 1 backup (due to time constraints we weren't able to implement a backup selection algorithm in case the Coordinator crashed, so we decided to allow for only 1 to exist at the time)

- When the coordinator dies, the backup simply create a new Coordinator initialized with the parameters from the backup class (making it so the newly created coordinator is updated with the last Coordinator's state before it crashed.

Recover from Worker's

How it works:

1. Every time we send a task to one of the workers, we store the work message on a map where the respective key is the worker's connection.
2. Using the *eof_received()* function of **Asyncio**, if a worker, dies we can detect it and we call the *redistributeWork()* function that allows us to fetch the last task we gave that worker, and give it to another.
3. In order to make the above idea work, the *redistributeWork()* function fetches the last message sent to that workrt (it's on the map referenced above) and puts it on a **queue** called *self.lostWork*.
4. Every time we'tr going to send more work to one of the workers, we first check if the *self.lostWork* queue has something.
    4.1 if True: we send the first message on the queue
    4.2 else: we send a new message with new work

Other considerations

- The way we were calling the map and reduce requests didn't take into consideration the eventuality that we might only have 1 blob. To circumvent this issue without re-writing our code we added a control flag self.singleBlob (set to True if we only have 1 blob in the datastore) to the coordinator. When we're about to present the final result we check if self.singleBlob is true, if it is we instead send a reduce reply and set self.singleBlob to false (so we don't enter an infinite loop).

- We implemented a MergeSort algorithm at the Worker side.

- Thanks to time constraints/scheduling with other subjects we weren't able to fully implement/bug fix everything to a state that we'd be happy to, but alas, this is the state that we managed to get to. Mostly everything works (with nothing crashing everything works wonders), the backup is able to generate a new coordinator with no problem on smaller texts and with 1 worker, but with multiple workers the final output, given when the coordinator crashes, turns out to be incorrect. Workers' fault tolerance was fully implemented and we believe it to be working perfectly however.

- Given more time we'd have changed the way we treat the breaking/receiving of messages (since we believe that to be the root cause of the big headache that the backup is giving us) – We'd adopt a strategy like, first sending the total size of the message in a separate message so that the receiver knows how large the incoming message is and builds a buffer inside a while loop.