Lab3: Eventual Consistency and Vector Clocks

Faiz Ahmed & Md Abu Noman Majumdar

Design

+ Used Vector clock
+ Important classes
  - **Server:** Handle all client requests and works as an entry point of all other objects.
  - **VectorClock**
  - **ClientDataProcessor**: Handles all clients data, always take data from "TempDataQueue" and after some processing, store the data inside "MessageQueueToPropagate"
  - **EventsProcessor**: Actually, handle data from other servers
  - **DistributedBoard**: Store the processed data to display
  - **DataPropagator**: Take the data from "MessageQueueToPropagate" and propagate data to other clients. If data propagation failed, it store that data inside another "MessageQueueToPropagate"
  - **DataResender**: Get the failed data from MessageQueueToPropagate (failed_message_queue_to_propagate") and retry to send all the failed data untill succeful result. DataResender also helps in *Network Segmentation* case. Because it always try to communicate with dead servers.

Task 1 – Why Vector Clocks ?

- **Logical clock** use just single timestamps (no local or global separetion) which cannot detect whether two events are causally related (One clock happened before second clock) or concurrent.

- But **Vector Clocks** use two separate timestamp one for global and another for local, which will help us distinguish between concurrent or sequential

Task 1 – Pros & Cons of Vector Clocks

- **Pros**:
  - Overcome the problem of distinguishing between causally related or concurrent events
- **Cons**
  - We need to send the entire Vector to each process for every message sent, in order to keep the vector clocks in sync. When there are a large number of processes this technique can become extremely expensive, as the vector sent is extremely large.

Task 1 – Vector Clocks Implementation

- Handle client event:
  - Incremented the self clock index value by 1

```
def increaseSelfClock(self):
    self.all_clocks[self.server_details.server_id - 1] += 1
```

- Handle event by other process:

```
for index in range(0, len(new_vector_clock)):
    self.all_clocks[index] = max(self.all_clocks[index], new_vector_clock[index])
    self.all_clocks[self.server_details.server_id - 1] += 1
    self.server_details.changeServerTitle(self.all_clocks)
```

Task 2 – Concurrent & Causal modifications

- Suppose we have three process p1, p2, p3
- Process p1 observed two events e2 (from p2), e3 from (p3) with VC2 = [1, 0, 0] and VC3 = [1, 0, 1]
  - e2 and e3 are causally related (e2 -> e3)
- Exampl2: VC2 = [1, 0, 0] and VC3 = [0, 0, 1]
  - e2 and e3 are concurrent
  - We used SERVER_ID to deals with concurrency. In this case e2 will execute first.

Task 3 – Network Segmentation Implementaion part

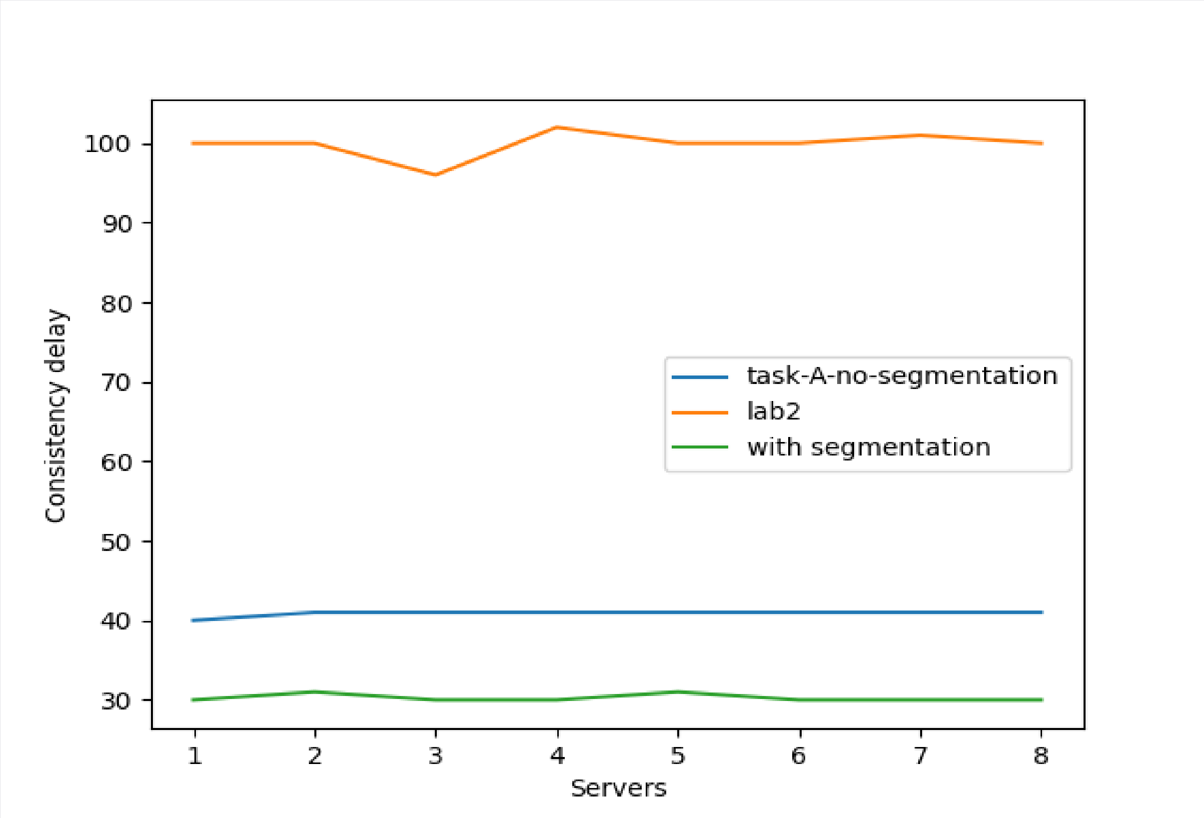
- "data_resender.py" class is responsible for maintaining consistency in Network Segmentation case
- This class always try to send the failed messages to the inactive servers.
- If a link gets up, that server can receive the message and can become consistent.

Task 4

+ Task A: Time required to reach consistency by servers:
  + No segmentation: [40,41,41,41,41,41,41,41]
+ Task B: Time required to reach consistency by servers:
  + Lab3: [40,41,41,41,41,41,41,41]
  + Lab2: [100,100,96,102,100,100,101,100]
+ Task C: Time required to reach consistency by servers:
  + With segmentation: [30,31,30,30,31,30,30,30]

  * all times are in seconds, and number of data 5 from each servers.

Task 4



100
90
80
70
60
50
40
30
Consistency delay
1
2
3
4
5
6
7
8
Servers
task-A-no-segmentation
lab2
with segmentation

Thank you !