Major tasks:
1. Extend the implementation of Branch.Withdraw and Branch.Deposit interface to enforce Monotonic Writes policy.
2. Extend the implementation of Branch.Withdraw and Branch.Deposit interface to enforce Ready your Writes policy.

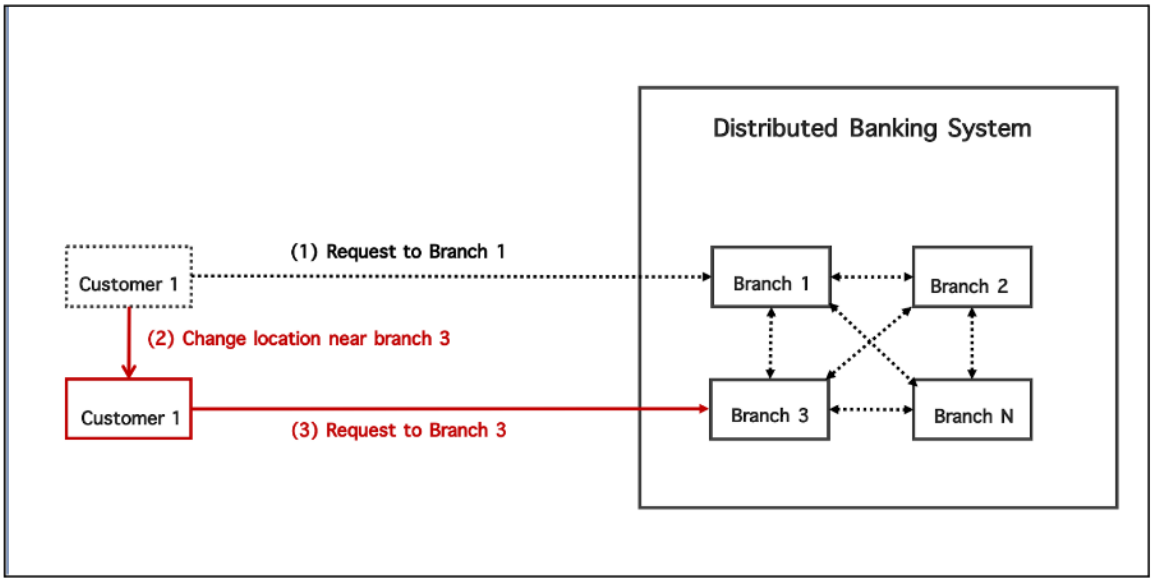

Diagram A: The customer changes the branches while submitting request to the Bank

1. Description
The customer described in (1) of diagram A, accesses the banking system by connecting to one of the replicas in a transparent way.

In other words, the application running on the customer's mobile device is unaware of which replica it is actually using. Assume the customer performs several update operations and then disconnects.

Later the customer accesses the banking system again possibly after removing to a different location or using a different access device. At that point the customer may be connected to a different replica than before as shown in (2), (3) of Diagram A.

However if the updates performed previously have not yet been propagated, the customer will notice inconsistent behavior. In particular, the customer would expect to see all previously made changes, but instead it appears as if nothing at all has happened.

This problem can be alleviated by introducing client-centric consistency. In essence, client-centric consistency provides guarantees for a single client concerning the consistency of accesses to a data store by that client.

1. Monotonic Writes
In many situations, it is important that write operations are propagated in the correct order to all copies of the data store. This property is expressed in monotonic-write consistency. In a monotonic-write consistency store, the following condition holds: A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

Suppose that a client has an empty bank account and deposits $100 in location A. Suppose that the customer also withdraws while in location B. How and when the deposit of $100 in location A will be transferred to location B is left unspecified. In this case, if the deposit request has not yet been received by the server in location B, then the withdrawal request will fail.

2. Read your Writes
A data store is said to provide read-your-writes consistency, if the following condition holds: The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

Suppose the customer deposits $100 in an empty bank account, and is planning to first check the bank account has $100 and then withdraw $100. The customer issues sequential order of requests (deposit -> query -> withdrawal) to the server. The customer may fail to withdraw the money from the bank because the query request can return $0.

This annoying problem arises because the query request contacted a replication which the new deposit request had not yet propagated.

2. Implementation
Unlike Project 1, where the customer communicates with only a specific branch with the same unique ID, Project 3 allows the customer to communicate with different branches for query, withdrawal, and deposit operations.

The Branch processes generate the IDs for write operations (deposit and withdraw) requested by the customer. The Customer process obtains the IDs of the write operations performed by the Branch processes. Both the Customer process and the Branch process maintain these sets of write IDs, i.e., writesets.

When the Customer sends a request to a Branch, it also sends its writeset. The Branch compares the received writeset to its own writeset, and uses them to enforce client-centric consistency in the banking system.

3. Input and Output

The input file contains one Customer and multiple Branch processes. The format of the input file follows Project 1, you will be using the destination parameter “dest” which has a value of a unique identifier of some branch.

```
[ // array of processes
  { // Customer process #1
    "id" : {a unique identifier of a customer or a branch},
    "type" : “customer”,
    "events" :  [{"interface" : {query | deposit | withdraw}, "money" : {an integer value}, “id“ :
{unique identifier of an event}, “dest” : {a unique identifier of the branch} }]
  }
  { // Branch process #1
    "id" : {a unique identifier of a customer or a branch},
    “type” : “branch”
    “balance” : {replica of the amount of money stored in the branch}
  }
  { … } // Branch process #2
  { … } // Branch process #3
  { … } // Branch process #N
]
```

Output file format for the Monotonic Writes

```
[ // array of customers
   { // Customer process #1
     “id” : { a unique identifier of a customer #1  }
      “balance” {the final result of the balance for customer #1}
   }
]
```

Output file format for the Read your Writes

```
[ // array of customers
   { // Customer process #1
     “id” : { a unique identifier of a customer #1  }
      “balance” : [the list of results of the read operations performed by customer #1]

   }
]
```

1. Monotonic Writes Example

The test-cases for the Monotonic Write consistency implementation will generate Customer processes that perform write operations on different Branch processes.

Example of the input file


```
[
 {
  "id" : 1,
  "type" : "customer",
  "events" :  [{"interface" : "deposit", "money" : 400, "id" : 1, "dest" : 1},{"interface" : "withdraw", "money" : 400, "id"
: 2, "dest" : 2 }, {"interface" : "query", "id" : 3, "dest" : 2 }]
 },
 {
  "id" : 1,
  "type" : "branch",
  "balance" : 0
 },
 {
  "id" : 2,
  "type" : "branch",
  "balance" : 0
 }
]
```


The customer will perform write operations to different Branch processes, and read the final result of the balance from the last Branch process that it requested the write operation. The balance of the customer should reflect the correct number of withdrawal and the deposits.

Expected output file:	**Wrong output file:**
[{"id": 1, "balance": 0 }]	[{"id" : 1, "balance" : 400 }]

2. Read your Writes Example

The test-cases for the Read your Writes consistency implementation will generate Customer processes that perform a sequence of write and read operations on different Branch processes.

Example of the input file


```
[
 {
   "id" : 1,
   "type" : "customer",
   "events" :  [{"interface":"deposit", "money":400, "id":1, "dest":1 }, {"interface":"query", "id": 2, "dest": 2  }, ]
 },


 {
   "id" : 1,
   "type" : "branch",
   "balance" : 0
 },
 {
   "id" : 2,
   "type" : "branch",
   "balance" : 0
 }
]
```


The customer will perform write and read operations to different Branch processes. The read operations performed by the customer should reflect the correct result of write operations that the customer performed before.

Expected output file:	**Wrong output file:**
[{"id": 1, "balance": 400 }]	[{"id": 1, "balance": 0 }]