The linked repository contains extensions to the cloned repository with the new following features:

1. Login and registration
2. Follow and unfollow
3. Reply to chirps
4. Pulling chirps from popular user, pushing chirps from unpopular user

I think the registration and login process is handled really well! While the pulling and pushing of new chirps is still under work. It is working but a better solution is still currently searched (since the current one is based on times).
Generally the system may appear quite slow because of a bug i couldn't address so far concerning the readBatch function (consequently, currently all the reading are just sequential).

The database is based on the following maps (for the following explanation the notation will be abused a little but i hope it is still perfectly understandable)

/** general purpose **/
Name2ids_ids_set *// set of ids which are associated with the same name (map's key)*
Id2names_names_set *// set of names which are associated with the same id (maps' key)*
id2Ids_mergedTo_register
id2Ids_lastPull_register
id2Ids_lastPush_register
id2Ids_followingFrom_register
/** user entity **/
user_email_register
user_password_register
user_time_register
user_writtenChirps_set
user_writtenReplays_set
user_followers_set
user_following_set
user_timeline_set
user_chirpIdToBePulled_set
user_chirpIdPushed_set
user_lastPullableChirp_register
user_lastPushedChirp_register
/** chirp entity **/
chirp_chirpId_register
chirp_userId_register
chirp_body_register
chirp_time_register
chirp_replays_set
/** replay entity **/
replay_replayId_register
replay_userId_register

replay_chirpId_register
replay_time_register
replay_body_register

So three entities are considered, user, chirp and replay. Their fields are the obvious ones and they are use to store a "record" of the entity. (the two fields user_lastPullableChirp_register, and user_lastPushedChirp_register and the general purpose maps will be discussed later).
Each entity is characterized by an ID used as primary key (each attribute of this entity is memorized in the correspondence map with key his ID, for example, the email can be gotten using antidote.map("user_password_register").register("my id").read()). The relation between users and chirps is 1 to N, while users to replays is 1 to N, and chirps to replays is 1 to N.

So, a user is uniquely identify by an id which is generated partially randomly but another attribute that should be unique and chosen by the user is his name which is used for example in the login as well.
This means we have to resolve conflicts with the procedure of registration and consequently of login when the db is geo replicated.
The strategy used to resolve this kind of problem is the following:

1. A user A (identified by idA) attempts to register with the username N.
    a. In the name2Ids map will be add to the set identified by N the value idA
    b. In the id2names map will be add to the set identified by idA the value N.
2. A user B (identified by idB) attempts to register with the same username N.
    a. If the server which is used by B already contains in the name2ids map, applied to N, a value (the set identified by the key N is not empty), then the request is rejected.
    b. If the server still does not have an id in the set name2Ids refered to the name N, then it completes the registrations
        i. name2Ids.set(N).add(IdB)
        ii. id2names.set(IdB).add(N)

With this strategy, a registration can always succeed (the point 2.a could be suppressed). The usage of set's is due to the wish of tracking the collisions without that two registrations with the same names can be destructive. Anyway, a state where one of the two set (name2Ids.set(N) or id2Names.set(someId)) have more than one entry, highlights a conflicts.

Now that we let the collision to occur, we have to solve it and this is done in the login process. Once the login is done the server pass the id to the client, and the client restlessly will perform all the request with the login credential and the provided id, so all the future request will be conflict free since they are performed providing an unambiguous id.

1. A user attempts to login with the name N.
    a. If the content of the name2Ids.set(N) contains no entry: Failure.
    b. If the content of the name2Ids.set(N) contains only one entry (let's say IdN) the name name is not replicated, so the login process can continue

   i. If the content of the id2names.set(IdN) contains one element, then no collision where store for this user. The password is verified and if it matchs with the waited one, the client is replied with IdN: success!
   ii. If the content of the id2names.set(IdN) contains two or more elements, the user will be asked to chose the name he wanna use, and then the registration is restarted from 1.b

c. If the content of the name2Ids.set(N) contains more than one entry, then two or more users are using the same name and only now the system detected it. The user will have to introduces his email, let's say M, and the list of user associated with that email (let's call this set S) is looked up, leading to the following situations
   i. If S contains only one element, then the db contains only one user with the email M, and we will call it IdC
      1. If IdC register before of any other user in the set name2Ids.set(N), then it will have the priority for the name he is using and the login continues from the point 1.b (considering the name free of conflicts).
      2. If IdC is not the oldest user in the db using that name, he will be asked to introduce a new user name and then, if the login process restarts
   ii. If S contains two or more entries, then two or more user registered with the same name and the same email. No possibility to distinguish them is possible, so the users will be consider as the same one, and the accounts will be "merged".
      1. The oldest id is identifier is detected: IdOld
         a. Foreach other id in S
            i. id2Ids_mergedTo_register.register(id).set(IdOld)

This strategy should address all the conflict situations. The one which i am more unsadisfied is the merging one, also because it will be require that each time an id is read, it is validated and reduced to the merged one (anyway this situation should be really uncommon).
The situation 1.b.ii may occur when a user repeat the disambiguation procedure twice in two different part of the word (in two times really close) inserting two different names.

Pulling and pushing
In the current commit the strategy is chosen randomly for testing purpose, in the coming commit the strategy will be restored to the one that should be.
The user U publishes a chirp C. Now the strategy of how the followers (indicated by the set user_followers_set.set(U), let's call it Fs) have to be acknowledged of the new chirp has to be decided.

1. If Fs has size smaller than a provided bound, the strategy is pushing
   a. U updates the time of user_lastPushedChirp_register with C.time, which indicates the time of the last chirp that was pushed.
   b. For each F in Fs the chirp will be pushed and control values updated
      i. id2Ids_lastPush_register with the combined key (U,F) is updated with C.time
      ii. F.timeline is updated by adding C

2. Otherwise the size of Fs is too big, and pushing is expensive: pulling is the strategy chosen and it will be let to the user to perform
    a. U updates the time of user_lastPullableChirp_register with C.time, which indicates the time when the last chirp was made available to be pulled.

When a user V wants to load his time line, it will consider the set of user it is following (indicated by user_following_set) and for each user W in this set it will perform both the following actions:

1. The values W's user_lastPullableChirp_register and (W,V)'s id2Ids_lastPull_register are compared.
    a. If the value is the same, no new chirps were added.
    b. If the values are different, then, or new chirps are available or just now it was detected that not all the chirps were added to the timeline.
        i. W's user_chirpIdToBePulled_set created after (W,V)'s id2Ids_followingFrom_register are added to the time line
        ii. (W,V)'s id2Ids_lastPull_register is updated with W's user_lastPullableChirp_register
2. The values W's user_lastPushedChirp_register and (W,V)'s id2Ids_lastPush_register are compared.
    a. If the value is the same, no new chirps were lost
    b. If the values are different, then, we are detecting that W published some chirps which V was not aware of .
        i. W's user_chirpIdPushed_set created after (W,V)'s id2Ids_followingFrom_register are added to the time line
        ii. (W,V)'s id2Ids_lastPush_register is updated with W's user_lastPushedChirp_register

Even though this solution does not convince me 100% i am quite confident it can handle concurrency, or at least i haven't figured out situations which may lead to losing chirps. Anyway, since this solution relays on times, it will be refactored using other strategies.

Cause of inefficiency
Currently i am using the following function

```
function antidoteReadBatch( arr ){
  // return antidote.readBatch( arr )
  // JUST FOR DEBUG
  return Promise.all(arr.map( (obj) => obj.read() ))
}
```

In the place of just antidote.readBatch. The parameter arr is an array of ***AntidoteObject***s and, at least as far as i have understood, this function should have the same result (except that for the efficiency "detail") that the correct antidote.readBatch(arr) call should have as the documentation reports:

- `readBatch(objects: AntidoteObject<any>[]): Promise<any[]>`

Anyway, using antidote.readBatch result in the following error, which i guess it may depends on some configurations

```
Error: BUFFER_SHORTAGE
    at DecodeBuffer.reserve
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\m
sgpack-lite\lib\flex-buffer.js:64:41)
    at uint8
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\m
sgpack-lite\lib\read-format.js:117:23)
    at Codec.decode
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\m
sgpack-lite\lib\read-core.js:22:16)
    at DecodeBuffer.fetch
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\m
sgpack-lite\lib\decode-buffer.js:26:21)
    at DecodeBuffer.read
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\m
sgpack-lite\lib\flex-buffer.js:166:28)
    at Object.decode
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\m
sgpack-lite\lib\decode.js:10:18)
    at ConnectionImpl.binaryToJs
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\a
ntidote_ts_client\dist\antidote.js:159:31)
    at CrdtMapImpl.binaryToJs
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\a
ntidote_ts_client\dist\antidote.js:593:28)
    at CrdtRegisterImpl.interpretReadResponse
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\a
ntidote_ts_client\dist\antidote.js:397:28)
    at ConnectionImpl.<anonymous>
(C:\Users\matti\Documents\Antidote\ProposedExtensions\node_modules\a
ntidote_ts_client\dist\antidote.js:219:38)
```