Homework Assignment 2 (12.5 points)

4DC3, Winter 2019

Due: March 4, 23:59pm

Marking

This assignment accounts for 12.5% of your final mark.

Instructions

Please submit your solution on Avenue as a single ZIP before the deadline. Only the last submitted version will be considered for grading. You are encouraged to use the `mix` tool to create a project directory (see the tutorial slides). Please see the course outline for the policy on late submissions.

1 Problem Statement

Distributed Bulletin Board:

A bulletin board consists of a list of *topics* $t_1, \ldots, t_k$ and *users* that can subscribe to specific topics and post new information about a topic. Newly posted information is forwarded to all currently subscribed users. The goal is to implement a distributed bulletin board system that supports the following operations:

`subscribe(`u_i`,`t_j`):` subscribes user u_i to topic t_j. This ensures that user u_i will be notified (by receiving a message) whenever an update is posted for topic t_j in the future. If the topic t_j does not exist yet, it is created with user u_i as the (only) subscribed user.

`unsubscribe(`u_i`,`t_j`):` unsubscribes user u_i from topic t_j such that u_i is no longer notified of updates to the topic.

`post(`u_i`,`t_j`,content):` user u_i posts the string `content` regarding topic t_j. This causes notification messages (including `content`) to be sent to all currently subscribed users.

Your implementation should adhere to the following guidelines:

- Initially, no user is subscribed to any topic.
- For each topic t_j, a *topic manager* is responsible for handling the subscription of users and implementing the broadcast of newly posted information to the current subscribers. To ensure fault-tolerance, a topic manager is replicated on all available nodes (described below). It is recommended that you implement this functionality in a module `TopicManager`. To be discoverable by users, the topic manager for topic t_j registers its process ID (PID) under the topic name in the process registry.

- Write a module `User` that provides an API to facilitate the interaction of users with the topic managers. The module should include the following functions:

  `subscribe(topic_name,user_name)`,
  `unsubscribe(topic_name,user_name)`,
  `post(user_name,topic_name,content)`,
  `fetch_news()`.

  A new user joins the system by executing `User.start(user_name)`, which registers the user's name (with this PID) in the global process registry.

- **Example:** Suppose that four nodes[1] `alpha@pc`, `beta@pc`, `gamma@pc`, and `delta@pc` are running and user Alice executes the following code on node `alpha@pc`:

  ```
  Node.connect(:"beta@pc")
  Node.connect(:"gamma@pc")
  Node.connect(:"delta@pc")

  User.subscribe("Alice","computing")
  :timer.sleep(5000)
  IO.puts("Updates to Alice's subscriptions: #{User.fetch_news()}")
  ```

  Meanwhile, Bob runs the following on node `beta@pc`:

  ```
  User.subscribe("Bob","computing")
  User.post("Bob","computing","Just found a new prime number.")
  User.unsubscribe("Bob","computing")
  ```

- When Alice executes `User.subscribe` for the topic "computing", a new topic manager process is spawned (on every connected node as described in item Fault-tolerance below), assuming this is the first time a user subscribes to "computing". Future subscriptions to computing by other users do not spawn new topic managers.

- Upon startup, the topic manager registers its PID in the process registry under the topic name and keeps running in the background where it handles user subscriptions and dissemination of posts. In the above code, Alice's process sleeps for 5 seconds while Bob posts a new message to the "computing" topic.

- When Alice's process resumes its operation and executes `User.fetch_news()`, she will receive the message previously posted by Bob, which was relayed to her by the topic manager of "computing".

- **Fault-tolerance:** We extend the above basic architecture to ensure that the system can survive the failure of nodes. To this end, a topic manager process is not only spawned on `alpha@pc` when Alice executes `subscribe` in the above example, but is also replicated on every other connected node, which in this case includes the other three nodes to which `alpha@pc` is connected, i.e., `beta@pc`, `gamma@pc`, `delta@pc`. Note that only one topic manager process should be registered under a topic name (in this case "computing") in the process registry at any point in time. This registered process is called *master topic manager* (for the topic "computing") and is responsible for handling subscription requests from the users and for broadcasting the content.

  The remaining topic manager instances, called *secondary managers*, merely monitor the health of each other's nodes (e.g., by using `Node.monitor`) and do not communicate with the users at this point.

  If a secondary manager notices that the node of the master topic manager process has crashed, one of the secondary managers takes up the role of the new master topic manager and registers itself in the

[1]Recall that you can start a new node with a specific name by running `iex --name "alpha"`.

process registry under the topic name. To avoid interrupting the availability of the service to the users, this transition requires all secondary topic managers to have stored a (reasonably) up to date list of subscribed users etc, which needs to be relayed to them by the master topic manager whenever its list is updated. To avoid running into impossibilities, we sacrifice strong consistency for availability, i.e., it is sufficient if your system makes a "best-effort" at keeping the states of the secondary topic managers consistent with the master topic manager's state.

Grading

1. **Architecture Description:** Describe the architecture and the design of your bulletin board implementation, including the types of messages that your processes use to communicate. (The writeup should be at most 1 page.)

   [Points: 2]

2. **Basic System:** Implement a distributed bulletin board according to the above requirements and guidelines when considering only a *single* node, i.e., all users execute their code as different processes on the same node. Note that in this case there is only one topic manager (per topic) and there is no need to replicate its state. Your code should be annotated using comments and structured appropriately into modules and functions.

   [Points: 7]

3. **Adding Fault-tolerance:** Extend your system design to take advantage of all currently connected nodes by replicating the topic manager as described in the above paragraph on fault-tolerance. You can assume that at most 1 node will fail by crashing, i.e., either the master topic manager or one of the secondary topic managers may stop functioning. You do *not* need to implement fault-tolerant consensus to achieve full marks. Instead, try to design a simple "best-effort" protocol that works in most cases for up to a single crash.

   [Points: 3.5]