CS321 Midsem

Berkeley Algorithm

Implementation in Prolog using Tartarus.

Anindya Vijayvargeeya, Roll 200101015

Overview

A distributed system consists of a set of processes and these processes communicate by exchanging messages. In distributed systems synchronization between processes is required for various purposes, for example in transaction processing and process control operations. For processes to be synchronized and have a common view of global time, clock synchronization algorithms are applied for ensuring that physically dispersed processes have a common knowledge of time.

Berkeley Algorithm

Berkeley algorithm is a method of clock synchronization in distributed computing which assumes no machine has an accurate time source. The algorithm starts with first (1) Leader election (one process would hereby be called the leader, others followers), then (2) the leader polls followers for their time, (3) the leader computes the average time, and finally (4) sends back to each follower their time adjustment values (how much they need to increase/decrease their time). In the final step, the value sent by the leader is not the timestamp the follower should set its clock to (the average time computed); rather, it is the relative adjustment value. This avoids further uncertainty due to RTT at the follower process.

This Implementation

Here are some details of this implementation of the Algorithm-

- The leader election is simulated by just making the leader jump around the platform in a ring topology.

- IPs and Ports used here to simulate machines of a distributed system are hardcoded, IP for all is localhost, and Ports vary from 6001 to 6005. Thus, we have 5 platforms across which time is synchronized here.
- All the code of the algorithm is in the `berkeley.pl` file. Files `plat{1-5}.pl` contains code to just start tartarus and consult this berkeley algo file. The berkeley algo can be started from any platform. Right now, it will run 5 times with the leader jumping around in cyclic order and then stops.

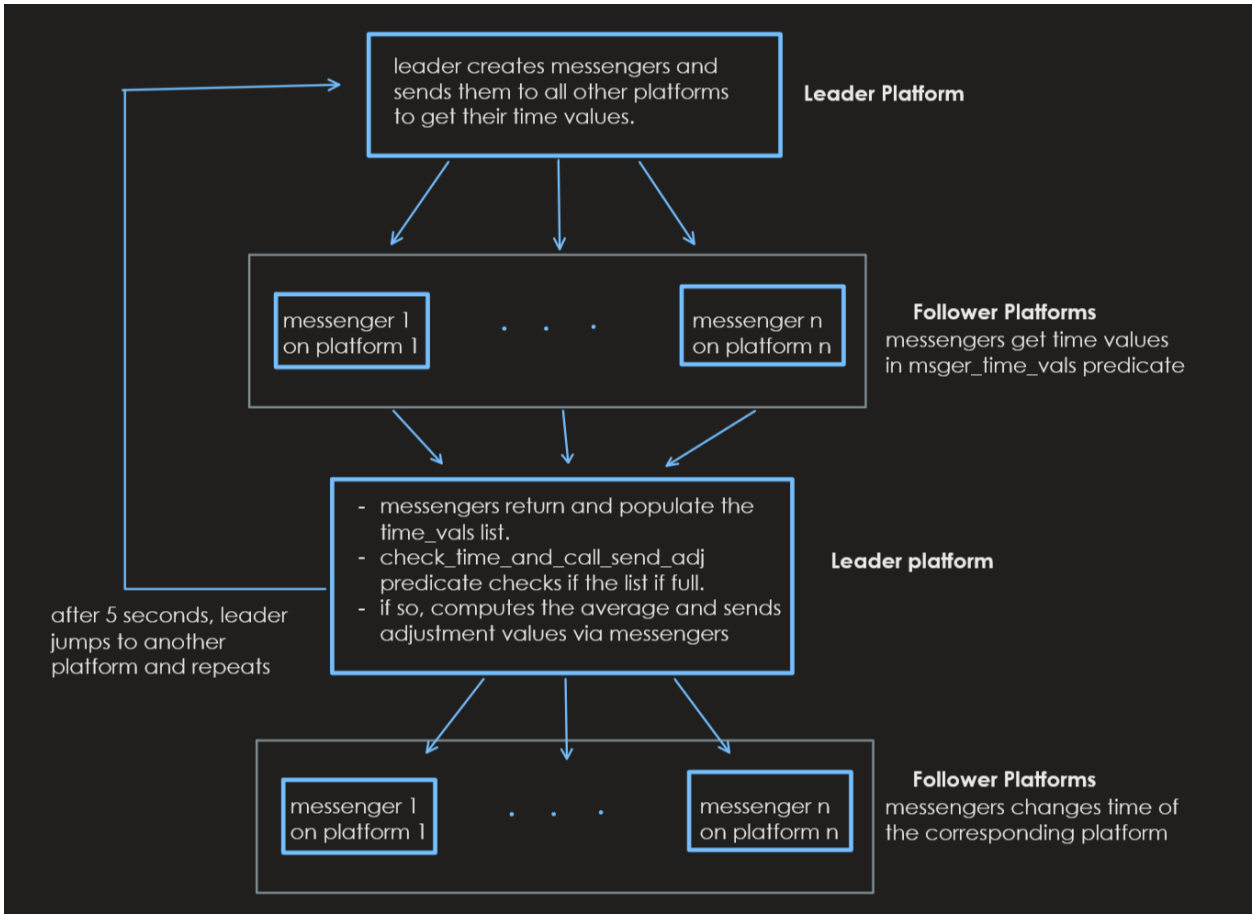
How this implementation simulates time

- Time values are stored as a dynamic predicate, available at each platform (code in `berkeley.pl` file) which can be accessed and modified using `get_time/1` and `set_time/1` predicates, respectively. Whenever `set_time/1` is called, it also prints out the time value that is set. Thus everytime time changes, the value is printed to console.
- To simulate time values drifting apart from one another (as in case of a distributed system), I've created a predicate, `drift_time/0`, that increases time by a random value, everytime it is called. When the platforms first starts, they run this predicate to initialize their clock value to a random integer close to 100.
- Everytime after correct time values are set, drift_time is called after a while to increment time with a random value.

The Synchronization Process

- A leader agent is created when the command `` `start_berkeley.` `` is run, at the platform where this is run. Thereby the leader does the synchronization process and moves to the next machine, in cyclic order.
- The synchronization process works as follows: First the leader creates messenger agents which goes to all other platforms, gets their time values and return back. On the leader platform they populate the list `` `time_vals/1` ``. When the list becomes full,i.e., leader knows all time values, it computes the average then sends time adjustment values to other platforms using messenger agents. When the messenger agents changes the time on the destination platform, they purge themselves.

Flowchart of the implementation



Predicates/Agents involved

Predicates for dealing with Time Values

- average/2
  - List is [10,12], average(Average, List). % sets Average to 11
  - Computes the average of the list. This is called with the time_vals to get the average time of all platforms.
- set_time/1
  - Sets the time of the platform to the specified time, and prints it out.
- get_time/1
  - Get the platform time.
- set_time_offset/1

    - Changes time by a given offset.
- drift_time/0
    - Increments time by a random value.

Leader Agent

Leader agent creates messenger agents and sends them to the follower platforms, and when these agents return with the timestamps of those platforms, the leader computes the average and sends the time adjustment values by creating new messenger agents and sends them back to each follower platform with the time adjustment value as their payload.

Payload predicates carried by the leader -

- all_platforms/2
    - Contains the list of (IP, Port) of all the platforms.
- jump_from_to/2
    - Provides the path to be followed by the leader agent
- iteration/2
    - Iteration number, each iteration is marked by leader jumping to a platform and starting polling, computing average time then finalling sending the time adjustment value.
    - Here we run the algorithm for 5 iterations only.

Other predicates used by the leader -

- time_vals/1
    - Stores a list of all the timestamps.
- msgers/1
    - Stores a list of messenger agent names along with corresponding platform's IP-Port values.

leader_handler/3

Resets variables and calls `dispatch_messengers` with the list of all platforms.

dispatch_messengers/2

If the platform information provided is the same as leader IP and Port, then it just adds the leader's timestamp in the `time_vals` list. Else, it creates a messenger and sends to the specified platform to get the time value of that platform. This information is stored in the `msgers/1` predicate, and the agents carry a payload of `leader_info/2` for them to know where to return back to.

check_time_and_call_send_adj/0

This predicate is called every time after `time_vals/1` predicate is updated. It checks where the list `time_vals` is full or not, and if so, computes the average and calls the `send_time_adjustment/2` for each platform. After waiting for 5 seconds, it moves the leader to a different platform to repeat the synchronization process.

send_time_adjustment/2

This predicate is called by the leader after it has done the computation of average time. It is called for each follower platform with the average time value, and a tuple denoting the messenger agent name (that got the timestamp) and the corresponding follower platform's details (IP and Port). The messenger is purged and a new messenger is created with a different handler `(messenger_handler2/3)` which takes the adjustment value and jumps to the corresponding follower platform.

Messengers

There are 2 types of messengers, having different handler functions.

- messenger_handler/3
    - This is for the messenger agents that get the timestamps from the follower processes.
    - An agent with this handler is created by the leader and is immediately sent to a follower platform, from where it gets the time value, saves it in `msger_time_value/2` (payload of this agent), and returns to the leader (the IP and Port of the leader is also stored as a payload, `leader_info/2` predicate. After returning to the leader platform, it updates the time_vals predicate.
- messenger_handler2/3
    - This is for the messenger agents that take the time adjustment values back to the follower processes.
    - An agent with this handler is created by the leader and is given the payload of the adjustment value in the `msger_time_value/2` predicate. Next it is moved to the follower platform, where first the agent adjusts time using `set_time_offset/1` and then after one second it increments time with a random value by calling `drift_time/0` predicate. This is to simulate errors in time values.

Execution Steps

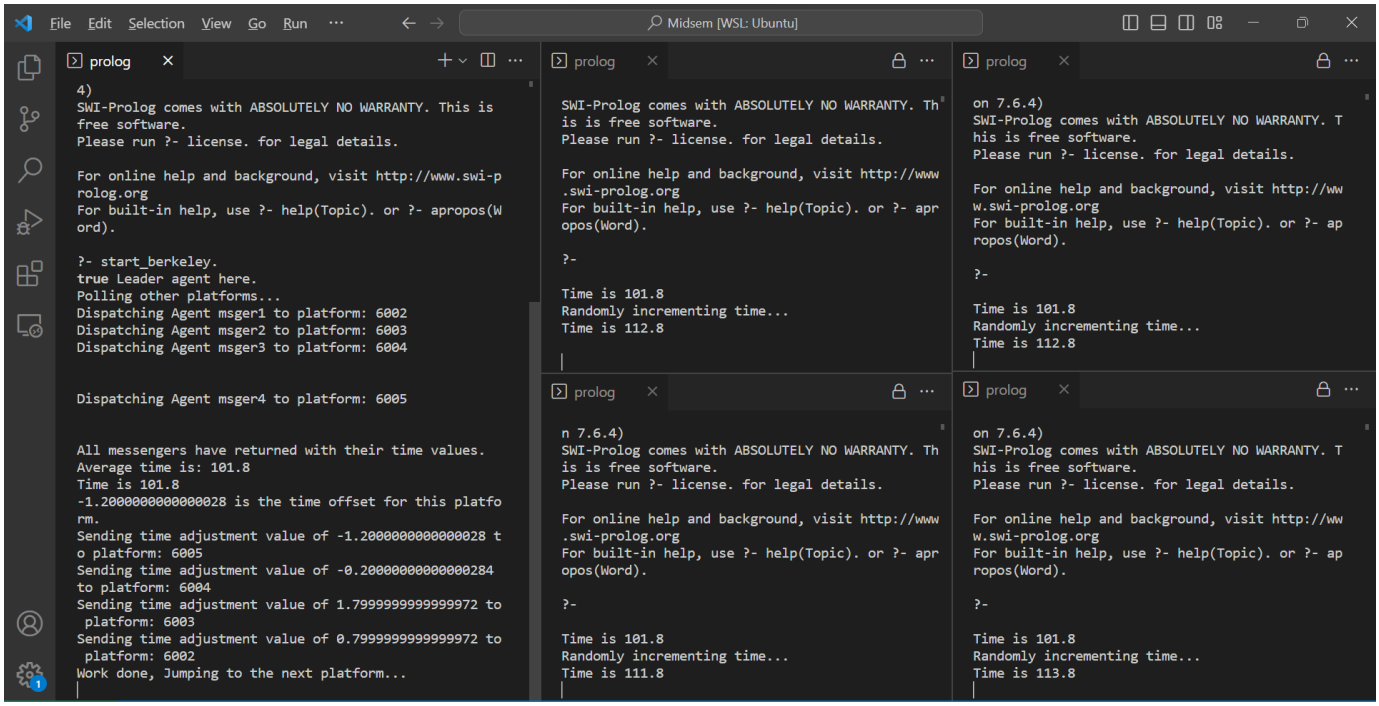
To run the code, open 5 terminals and run

```
prolog plat{x}.pl
```

with `{x}` replaced by numbers 1-5. All these 5 files consult `tartarus.pl` and `berkeley.pl`, so all of them must be present in the same folder. Then on any of these terminals, run

```
?- start_berkeley.
```

This creates the leader agent (on the platform from which this command is executed), which starts the synchronization process, as every time it completes the process on one platform, it jumps to the next platform.



Screenshot of running the algorithm for one iteration.

Future Work

These are some of the improvements that could be made over this implementation -

Better way to simulate time

Here I just encoded a global variable whose value tells the time and is changed randomly after some time intervals, rather than changing the time deterministically at different rates with respect to time, as in the real systems.

Not using hardcoded IPs and Ports

The IPs and Ports are hardcoded in this implementation. Rather, to simulate an actual distributed system, future implementations could provide options to machines to dynamically join and leave, as in a P2P network. The nodes currently in the system will be used to average the time.

Leader Election

The Leader in this algorithm doesn't just go around in circles; rather it is elected through some leader election algorithm. This could be a possible improvement over this implementation where the path taken by the leader is hardcoded.