CS751: Principles of Concurrent and Parallel Programming: Project

Tezan Sahu [170100035], Akash Kumar [160020009]

November 2019

Berkeley Clock Synchronization

This project involves a `python` implementation of the *Clock Synchronization algorithm used by TEMPO in Berkeley UNIX*, based on the work of *Riccardo Gusella* & *Stefano Zatti*.

Our implementation of the Berkeley Clock Synchronization algorithm is a slight variant of the original work and can be found here.

1 Background

The advantages of the distributed systems are so attractive that there is a gradual shift from the centralized system era to the distributed systems era. But, in a distributed system, the different nodes maintain their own time using local clocks and their time values may not be same for the different nodes, i.e. there is no global clock within the system so that that the various activities in the distributed environment can be synchronized.

Clock synchronization is a method of synchronizing clock values of any two nodes in a distributed system with the use of external reference clock or internal clock value of the node.

2 Problem Statement

The variant of the Berkeley Clock Synchronization algorithm implemented by us tries to solve the issue caused in distributed environments due to different unsynchronized local clocks by providing a fault-tolerant averaging of time differences.

Our system contains a master node and a set of slave nodes that periodically query master node to get the synchronized time for the system.

3 Implemented Algorithm

1. An individual node is chosen as the **master node** from a pool of nodes in the network. This node is the main node in the network which acts as a master and rest of the nodes act as **slaves**.

2. Periodically, the master queries all of the slave nodes for their local time and each node replies with their current time.

3. The master then computes the time difference between its local clock and that of each of the slaves [δ values for each slave].

4. The master computes a **fault tolerant average** of the δ values: The master selects the largest set of slaves that do not differ from each other by more than a small quantity (γ) & averages these δ values.

5. The master asks each slave to correct their respective clocks by sending them the synchronized time obtained by using the fault-tolerant average δ.

6. This synchronization procedure is repeated by the master every *T* seconds.

4 Assumptions during the Implementation

- The master node does not fail, i.e., the election algorithm to choose a new master node in case of failure is not implemented
- The set of slave nodes (alive and failed) does not change

5 Features that our Implementation Offers

- **Process-local clocks** for each node: Each node in our system owns a local clock [implemented in the `Clock` class in `clock.py`] that ticks every millisecond & can be set with the synchronized time.
- Synchronization rounds take place periodically at ***T* = 5 seconds**.
- The master node also synchronizes its time along with the slaves [similar to that mentioned in the paper]
- As long as a slave is alive its time is **eventually synchronized**
- Our implementation uses **UDP-based sockets** for messaging, which speeds up communication because it offers no deliverability guarantees.
- **Robustness to message delay:** The threshold (γ) for δ values of each slave guarantee that delayed messages from slaves corresponding to previous rounds are ignored while calculating the average. Moreover, the data structure containing the details of slaves is flushed at the end of each synchronization round. In our implementation, γ **= 2 seconds**
- **Robustness to slave halting failures:** The master continues with the synchronization algorithm as long as there is at least slave that sends its time for a given synchronization round.

- For testing purposes, the Clock class is implemented such that it can use a **user-defined drift rate** while ticking. This is implemented in the slave nodes by an argument `drift_rate` passed while starting a slave node from the CLI.

  Note: `drift_rate` *> 1 indicates a faster clock while* `drift_rate` *< 1 indicates slower clock.*

- Slave nodes print the times of their local clock immediately before & after synchronization from the master node to the console.

6 Running the Project

1. Clone this repository using git clone and `cd` into it.
2. To start the master node, type `python3 master_udp.py`. This would trigger the master node, which will wait for a slave to connect so that the synchronization process could start.
3. Fire up new terminal(s) and trigger the slave node(s) in them by typing `python3 slave_udp.py -drift_rate <desired_drift_rate>`. Not specifying the argument will set the `drift_rate` to 1 by default.