ID 2201 Report

Loggy: A Logical Time Logger

Yilai Chen

Introduction

The task is to implement a logging procedure that receives log events from a set of workers. The events are tagged with the Lamport time stamp of the worker, and the events must be ordered before being written to stdout.

Construction

This project implements a distributed system with logical clocks. The `mlogger` module manages the logging of events with time synchronization, ensuring message order. The `worker` module simulates worker nodes that exchange messages, updating their logical clocks based on message timestamps. The `test` module orchestrates the interaction between four worker nodes, setting their peers, starting the logger, and then running for a fixed duration before stopping the system. The system aims to demonstrate how logical clocks maintain the correct sequence of events across distributed nodes.

Output 1

Initial Output:

```
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
log: na john {received,{hello,77}}
log: na paul {sending,{hello,68}}
....
```

It can be seen that without a logical clock, the log often shows that the information is received before it is sent, causing confusion in the log.

Time module

```
zero() ->
  0.
```

Initial time as 0.

```
inc(_, T) ->
  T+1.
```

Add 1 to original time.

```
merge(Ti, Tj) ->
  case leq(Ti, Tj) of
    true ->
      Tj;
    false ->
      Ti
  end.

leq(Ti, Tj) -> % received node's time, msg's time
  Ti =< Tj.
```

Return the bigger(later) time. Used to reset the Node's time when receiving message.

```
clock(Nodes) ->
  lists:map(fun(Node) -> {Node, 0} end, Nodes).
```

Initialize the clocks of all nodes in the distributed system. It accepts a list of nodes Nodes and creates a tuple {Node, 0} for each node, indicating that the initial value of the node's clock is 0. The return value is a list containing a tuple of each node and its initial clock value.

e.g. [{node1, 0}, {node2, 0}, {node3, 0}]

```
update(Node, Time, Clock) ->
  lists:keyreplace(Node, 1, Clock, {Node, Time}).
```

Find a node named Node in list Clock, and update it as {Node, Time}.

```
safe(Time, Clock) ->
  lists:map(fun({_, LatestT}) -> leq(Time, LatestT) end, Clock).
```

Check whether Time is less than or equal to the latest timestamp LatestT of each node.

Output 2

```
5> test:run(1000,100).
log: 1 john {sending,{hello,57}}
log: 1 paul {sending,{hello,68}}
log: 2 paul {sending,{hello,20}}
log: 2 ringo {received,{hello,57}}
log: 3 paul {sending,{hello,16}}
log: 3 ringo {sending,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 4 john {received,{hello,77}}
log: 5 john {received,{hello,20}}
log: 5 ringo {sending,{hello,20}}
log: 6 ringo {sending,{hello,97}}
log: 6 george {received,{hello,20}}
log: 6 john {sending,{hello,84}}
log: 7 george {received,{hello,84}}
log: 7 john {sending,{hello,7}}
log: 8 john {sending,{hello,23}}
log: 8 george {received,{hello,16}}
log: 8 paul {received,{hello,7}}
```

```
log: 9 paul {sending,{hello,60}}
log: 9 george {received,{hello,97}}
log: 10 paul {sending,{hello,79}}
log: 10 george {sending,{hello,100}}
log: 11 paul {sending,{hello,33}}
log: 11 george {received,{hello,23}}
log: 11 ringo {received,{hello,100}}
log: 12 paul {sending,{hello,39}}
log: 12 ringo {sending,{hello,20}}
log: 13 ringo {received,{hello,60}}
log: 13 george {received,{hello,20}}
log: 14 george {sending,{hello,40}}
log: 15 john {received,{hello,40}}
log: 16 john {sending,{hello,56}}
log: 17 ringo {received,{hello,56}}
log: 17 john {sending,{hello,18}}
log: 18 ringo {received,{hello,79}}
log: 18 george {received,{hello,18}}
log: 19 ringo {received,{hello,33}}
log: 19 george {sending,{hello,95}}
log: 20 ringo {sending,{hello,90}}
log: 20 john {received,{hello,95}}
log: 20 george {received,{hello,39}}
log: 21 paul {received,{hello,90}}
log: 21 john {sending,{hello,29}}
log: 22 george {received,{hello,29}}
log: 22 john {sending,{hello,96}}
log: 23 george {sending,{hello,10}}
log: 24 george {received,{hello,96}}
log: 24 paul {received,{hello,10}}
log: 25 paul {sending,{hello,90}}
log: 25 george {sending,{hello,53}}
log: 26 paul {sending,{hello,85}}
log: 26 george {sending,{hello,65}}
log: 26 john {received,{hello,53}}
queue remains: 11
stop
```

Answer of Questions in Requirements

1. **Did you detect entries out of order in the first implementation, and if so, how did you detect them?**

   Just check the logs and find that "Sending N" happens after somebody "Receiving N". Out of order.

2. **What is it that the final log tells us?**

   It logs when a message is sent by one node and when it is received by another node.

3. **Did events happen in the order presented by the log?**

   Yes, the events should happen in the order presented by the log according to the logical clock, even if the physical order of events might differ.

4. **How large will the holdback queue be, make some tests and try to find the maximum number of entries.**

   Sleep = 1000, jitter = 100, Max Holdback Queue Length = 14.2

   Sleep = 1000, jitter = 50, Max Holdback Queue Length = 12.8

   Sleep = 1000, jitter = 10, Max Holdback Queue Length = 10.0

   Sleep = 100, jitter = 100, Max Holdback Queue Length = 26.8

   Sleep = 100, jitter = 50, Max Holdback Queue Length = 24.8

   Sleep = 100, jitter = 10, Max Holdback Queue Length = 20.4

   Sleep = 10, jitter = 100, Max Holdback Queue Length = 32.2(38)

   Sleep = 10, jitter = 50, Max Holdback Queue Length = 31.8

   Sleep = 10, jitter = 10, Max Holdback Queue Length = 30.6 (37)

	jitter = 100	**jitter = 50**	**jitter = 10**
Sleep = 1000	14.2	12.8	10.0
Sleep = 100	26.8	24.8	20.4
Sleep = 10	32.2(38)	31.8	30.6

Conclusion:

- In the state of four nodes, the sleep time and jitter value determine the Holdback Queue Length.
- Under the same sleep conditions, the greater the jitter, the greater the confusion of the message sequence, resulting in a larger Holdback Queue Length.
- Under the same jitter conditions, the smaller the sleep, the larger the Holdback Queue Length, where the maximum value appears when Sleep = 10, jitter = 100, largest length is 38.

Vector

	jitter = 100	**jitter = 50**	**jitter = 10**
Sleep = 1000	63.3	61.8	62.0
Sleep = 100	322.4	392.2	482.0
Sleep = 10	562.4(571)	927.4(952)	2234.6(2440)

Comparison with Lamport clock:

Vector clock can capture the causality of all nodes, while Lamport clock can only partially capture causality. Therefore, the holding queue of vector clock is longer, especially under high concurrency and low latency jitter.
Lamport clock holding queue is shorter: Under the same Sleep and Jitter conditions, the holding queue length of Lamport clock is relatively short (for example, under Sleep = 10, Jitter = 100, Lamport clock holding queue is 32.2, while vector clock is 562.4).