Tribhuvan University
Institute of Engineering
Pulchowk Campus

Distributed Systems
Lab 5:
Implementation of Lamport's Timestamp

Submitted by:

Aayush Lamichhane
(075BCT004)

Bishal Katuwal
(075BCT028)

Submitted to:
Department of Electronics and Computer Engineering,
Pulchowk Campus

Submitted on:
August 9, 2022

Title

Implementation of Lamport's Timestamp

Basic Theory

Problem of clock synchronization

Distributed systems lack a global clock. All the processes have their own local clock, but due to clock skew and clock drift they have no direct way to know if their clock is in check with the local clocks of the other processes in the system. This problem is referred to as the problem of clock synchronization.

To solve the problem of clock synchronization, a central time server (Cristian's Algorithm) can be used. The problem with a central time server is that its error depends on the round-trip time of the message from process to time server and back.Thu, a mechanism called a logical clock was invented.

Logical clock

Logical clocks are mechanisms based on capturing chronological and causal relationships of processes and ordering events based on these relationships. The first implementation, the Lamport timestamps, was proposed by Leslie Lamport in 1978 and still forms the foundation of almost all logical clocks.

Lamport Timestamps algorithm

A Lamport logical clock is an incrementing counter maintained in each process. Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender (causality). The algorithm of Lamport Timestamps can be captured in following rules:

1. All the process counters start with value 0.
2. A process increments its counter for each event (internal event, message sending, message receiving) in that process.
3. When a process sends a message, it includes its (incremented) counter value with the message.
4. On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one.

Looking at these rules, we can see the algorithm will create a minimum overhead, since the counter consists of just one integer value and the messaging piggybacks on inter-process messages.

One of the shortcomings of Lamport Timestamps is rooted in the fact that they only partially order events (as opposed to total order). Partial order indicates that not every pair of events need be comparable. If two events can't be compared, we call these events concurrent. The problem with Lamport Timestamps is that they can't tell if events are concurrent or not. This problem is solved by Vector Clocks.

Implementation

Pseudocode

```
# event is known

time = time + 1;

# event happens

send(message, time);
```

The algorithm for receiving a message is:

```
(message, time_stamp) = receive();
time = max(time_stamp, time) + 1;
```

Code

```
from flask import Flask, request, jsonify
import sys
import requests

app = Flask(__name__)
port_number = int(sys.argv[1])
assert port_number


own_timestamp = 1


def process_message(message):
   print("message recieved ", message, "at timestamp ",
own_timestamp)
```

```
def receive_handler(message, timestamp):
    global own_timestamp
    own_timestamp = max(own_timestamp, timestamp)
    process_message(message)


def trigger_event():
    global own_timestamp
    own_timestamp += 1
    print("Event triggered. New timestamp: ", own_timestamp)


def send_handler(message, address):
    trigger_event()
    send_via_channel(message, address)


def send_via_channel(message, to_address):
    requests.get(
        f"http://localhost:{to_address}/receive",
        params={"message": message, "timestamp": own_timestamp},
    )
    print("message sent ", message, "at timestamp ",
own_timestamp)


@app.route("/receive", methods=["GET"])
def receive():
    message = request.args.get("message")
    timestamp = int(request.args.get("timestamp"))
    receive_handler(message, timestamp)
    return jsonify({"Response": "OK"}), 200


@app.route("/send", methods=["GET"])
def send():
    message = request.args.get("message")
    address = request.args.get("address")
    send_handler(message, address)
    return jsonify({"Response": "OK"}), 200
```

```
@app.route("/event", methods=["GET"])
def event():
   trigger_event()
   return jsonify({"Response": "OK"}), 200


app.run(port=port_number)
```

Here multiple flask processes are run using `python node.py address(port`

Then, each process can call the trigger_event() whenever an event occurs. When a message is sent the trigger_event is automatically called. When a process receives a message, the message's timestamp and its own timestamp is used to update the node's timestamp. Whichever is higher is used.

Here is a sample run,

Using 2 processes running on port 5000 and 5001, when we GET at the following

http://127.0.0.1:5000/send?message=hello&address=5001

The process at 5000 sends a message to the process at 5001 and the receiving message uses the timestamp of the process at 5000 to update its own timestamp.

Process at 5000

```
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
^[[AEvent triggered. New timestamp:  2
message sent  hello at timestamp  2
127.0.0.1 - - [09/Aug/2022 23:05:32] "GET /send?message=hello&address=5001
HTTP/1.1" 200 -
Event triggered. New timestamp:  3
127.0.0.1 - - [09/Aug/2022 23:05:58] "GET /event HTTP/1.1" 200 -
Event triggered. New timestamp:  4
127.0.0.1 - - [09/Aug/2022 23:05:59] "GET /event HTTP/1.1" 200 -
Event triggered. New timestamp:  5
127.0.0.1 - - [09/Aug/2022 23:06:00] "GET /event HTTP/1.1" 200 -
Event triggered. New timestamp:  6
message sent  hello at timestamp  6
127.0.0.1 - - [09/Aug/2022 23:06:07] "GET /send?message=hello&address=5001
HTTP/1.1" 200 -
```

Process at 5001

```
 * Serving Flask app node
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5001
Press CTRL+C to quit
message recieved  hello at timestamp  2
127.0.0.1 - - [09/Aug/2022 23:05:32] "GET /receive?message=hello&timestamp=2
HTTP/1.1" 200 -
message recieved  hello at timestamp  6
127.0.0.1 - - [09/Aug/2022 23:06:07] "GET /receive?message=hello&timestamp=6
HTTP/1.1" 200 -
```

Conclusion

In this way ”Lab5 :Implementation of Lamport’s Timestamp” was completed.