ID2201 HT23 Distributed Systems, Basic Course (50929)

Report Homework 3 - Loggy - a logical time logger

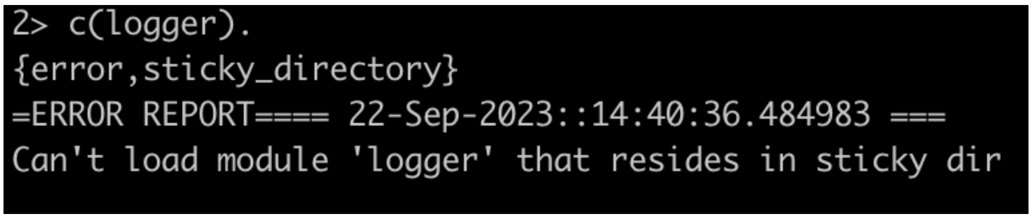
by Eugen Lucchiari Hartz

September 27, 2023

1 Introduction

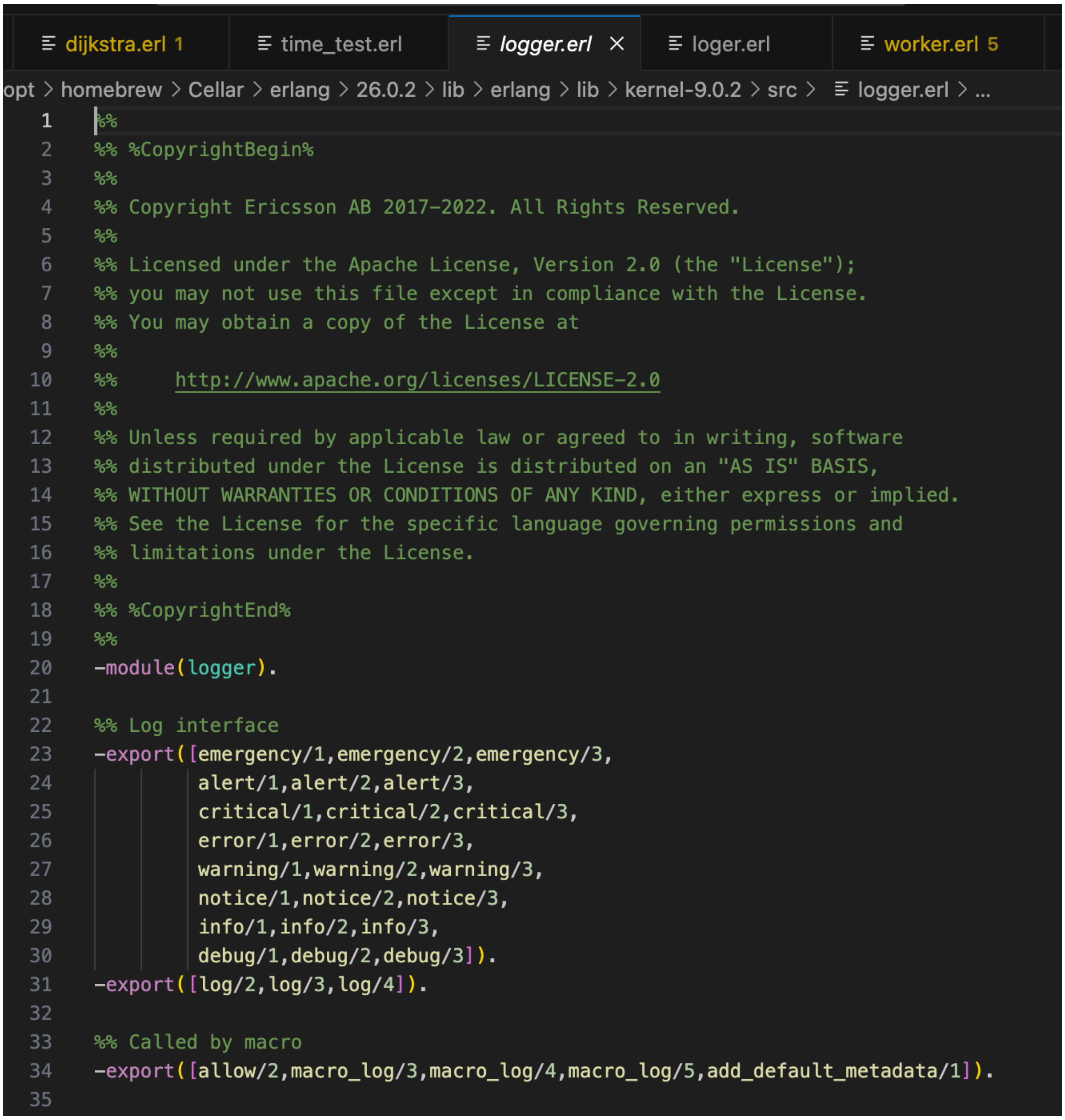
In this homework a logical time logger was developed using Erlang. Accordingly, this assignment was about how to use logical time in a practical example. The aim was to o implement a logging procedure that receives log events from a set of workers. The events are tagged with the Lamport time stamp of the worker, and the events must be ordered before being printed.
Working with logical time is an important aspect of Distributed Systems as it makes it possible to order events and establish a consistent timeline across multiple nodes. Accordingly, causality relationships can be detected which ensures proper synchronization and reliable communication among nodes.

2 Main problems and solutions



```
2> c(logger).
{error,sticky_directory}
=ERROR REPORT==== 22-Sep-2023::14:40:36.484983 ===
Can't load module 'logger' that resides in sticky dir
```

The file is in the correct directory which makes this issue a bit strange. It also only happened with the logger.erl file. After renaming the logger file to another random name like loger.erl it suddenly worked to compile the file. The reason for this issue from what I discovered is that in the Erlang environment there is already an existing module with the name logger. I found that out because when I clicked on logger within the test file while pressing command it redirected me to an already existing logger module. Accordingly, I assume that this causes the above displayed error message.


dijkstra.erl 1
time_test.erl
logger.erl
loger.erl
worker.erl 5
opt > homebrew > Cellar > erlang > 26.0.2 > lib > erlang > lib > kernel-9.0.2 > src > logger.erl > ...

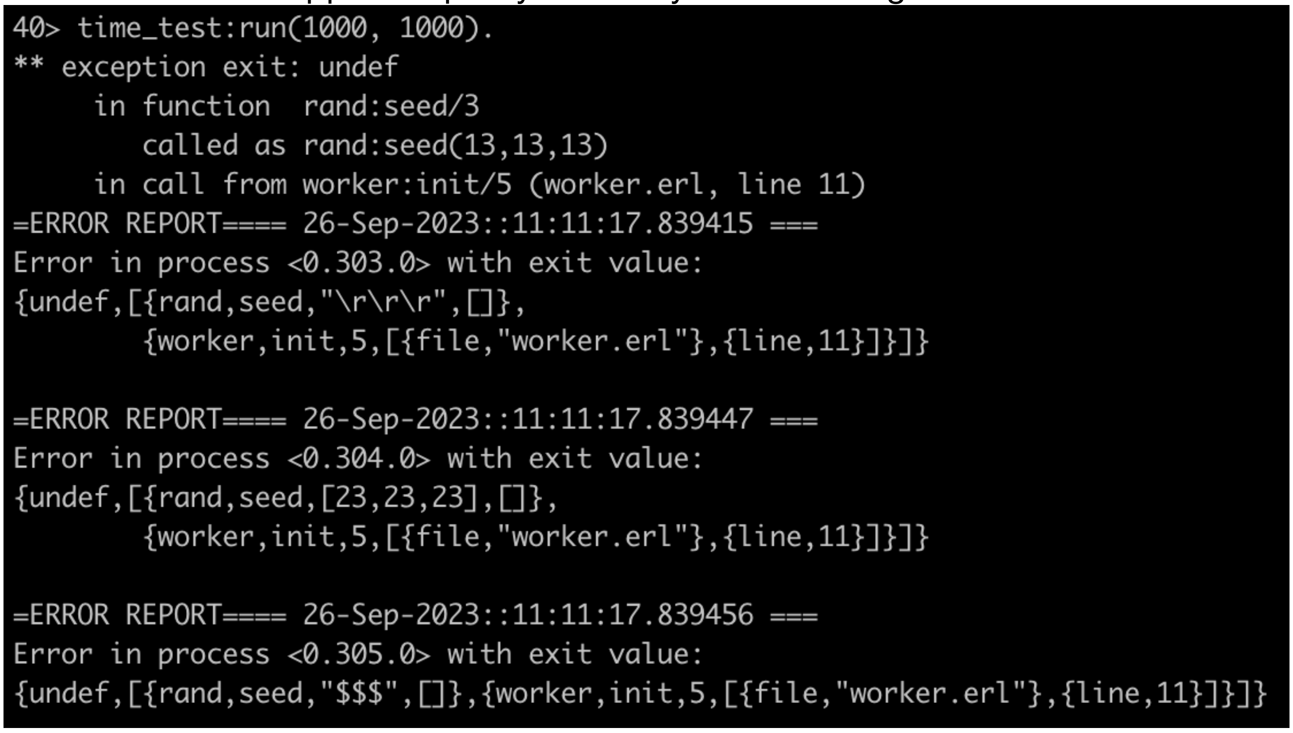
```erlang
%%
%% %CopyrightBegin%
%%
%% Copyright Ericsson AB 2017-2022. All Rights Reserved.
%%
%% Licensed under the Apache License, Version 2.0 (the "License");
%% you may not use this file except in compliance with the License.
%% You may obtain a copy of the License at
%%
%%     http://www.apache.org/licenses/LICENSE-2.0
%%
%% Unless required by applicable law or agreed to in writing, software
%% distributed under the License is distributed on an "AS IS" BASIS,
%% WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
%% See the License for the specific language governing permissions and
%% limitations under the License.
%%
%% %CopyrightEnd%
%%
-module(logger).

%% Log interface
-export([emergency/1,emergency/2,emergency/3,
         alert/1,alert/2,alert/3,
         critical/1,critical/2,critical/3,
         error/1,error/2,error/3,
         warning/1,warning/2,warning/3,
         notice/1,notice/2,notice/3,
         info/1,info/2,info/3,
         debug/1,debug/2,debug/3]).
-export([log/2,log/3,log/4]).

%% Called by macro
-export([allow/2,macro_log/3,macro_log/4,macro_log/5,add_default_metadata/1]).

```

Another error that appeared pretty randomly when running the test is this one:



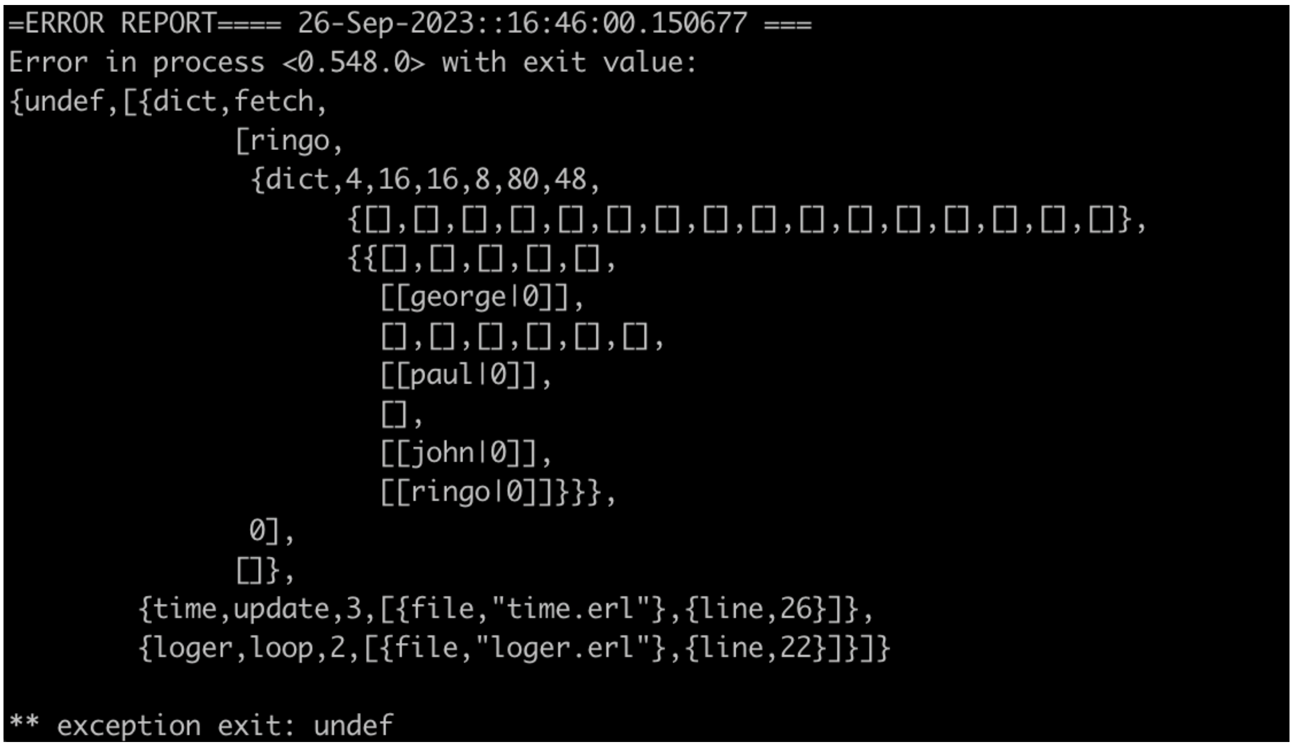
```
40> time_test:run(1000, 1000).
** exception exit: undef
     in function  rand:seed/3
        called as rand:seed(13,13,13)
     in call from worker:init/5 (worker.erl, line 11)
=ERROR REPORT==== 26-Sep-2023::11:11:17.839415 ===
Error in process <0.303.0> with exit value:
{undef,[{rand,seed,"\r\r\r",[]},
        {worker,init,5,[{file,"worker.erl"},{line,11}]}]}

=ERROR REPORT==== 26-Sep-2023::11:11:17.839447 ===
Error in process <0.304.0> with exit value:
{undef,[{rand,seed,[23,23,23],[]},
        {worker,init,5,[{file,"worker.erl"},{line,11}]}]}

=ERROR REPORT==== 26-Sep-2023::11:11:17.839456 ===
Error in process <0.305.0> with exit value:
{undef,[{rand,seed,"$$$",[]},{worker,init,5,[{file,"worker.erl"},{line,11}]}]}
```

could be fixed by changing “rand” back to “random” even if you get the warnings which can be ignored

using a list instead of the dict module to represent the clock because the use of dict causes an error.



```
=ERROR REPORT==== 26-Sep-2023::16:46:00.150677 ===
Error in process <0.548.0> with exit value:
{undef,[{dict,fetch,
              [ringo,
               {dict,4,16,16,8,80,48,
                     {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
                     {{[],[],[],[],[],
                       [[george|0]],
                       [],[],[],[],[],[],
                       [[paul|0]],
                       [],
                       [[john|0]],
                       [[ringo|0]]}}},
               0],
              []},
        {time,update,3,[{file,"time.erl"},{line,26}]},
        {loger,loop,2,[{file,"loger.erl"},{line,22}]}]}

** exception exit: undef
```

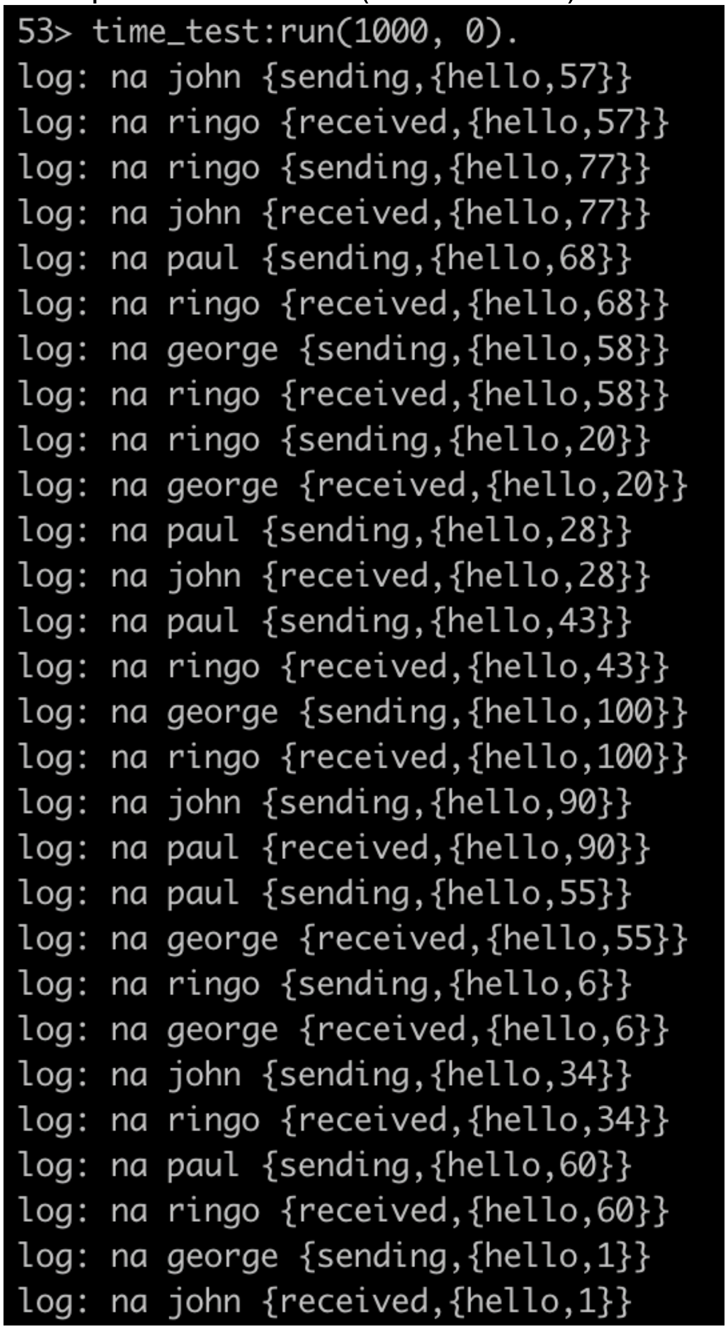
3 Evaluation

Run some tests and try to find log messages that are printed in the wrong order. How do you know that they are printed in the wrong order? Experiment with the jitter and see if you can increase or decrease (eliminate?) the number of wrong entries.

When running the test file for example like this time_test:run(10, 100). The first value, so the sleep parameter, represents the amount of time (in milliseconds) that each worker process should sleep before sending a message to the logger. The second argument, so the Jitter introduces randomness into the sleep times.

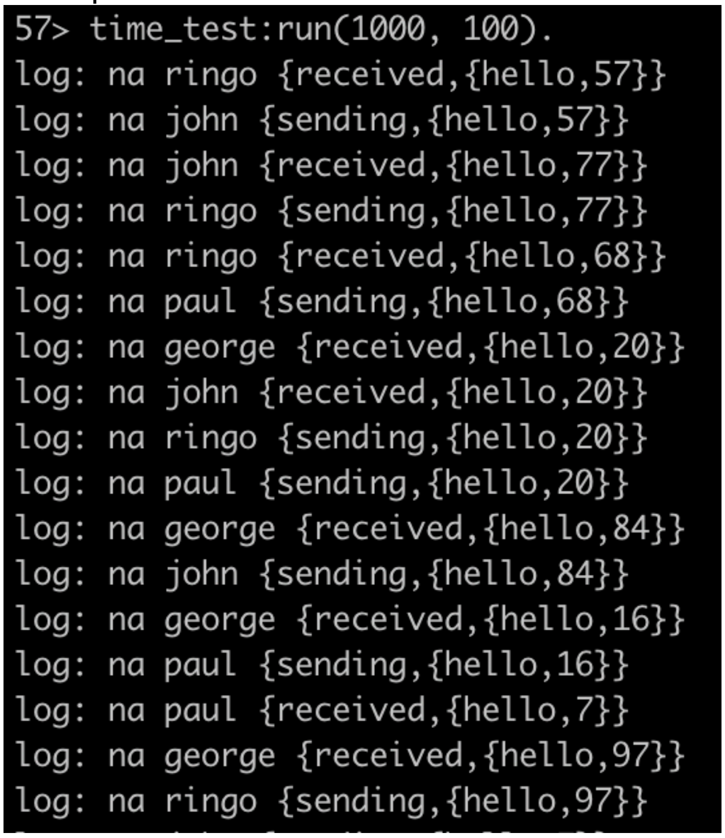
In order to analyze the influence of jitter, some tests are performed below:

Example without Jitter (correct order)



```
53> time_test:run(1000, 0).
log: na john {sending,{hello,57}}
log: na ringo {received,{hello,57}}
log: na ringo {sending,{hello,77}}
log: na john {received,{hello,77}}
log: na paul {sending,{hello,68}}
log: na ringo {received,{hello,68}}
log: na george {sending,{hello,58}}
log: na ringo {received,{hello,58}}
log: na ringo {sending,{hello,20}}
log: na george {received,{hello,20}}
log: na paul {sending,{hello,28}}
log: na john {received,{hello,28}}
log: na paul {sending,{hello,43}}
log: na ringo {received,{hello,43}}
log: na george {sending,{hello,100}}
log: na ringo {received,{hello,100}}
log: na john {sending,{hello,90}}
log: na paul {received,{hello,90}}
log: na paul {sending,{hello,55}}
log: na george {received,{hello,55}}
log: na ringo {sending,{hello,6}}
log: na george {received,{hello,6}}
log: na john {sending,{hello,34}}
log: na ringo {received,{hello,34}}
log: na paul {sending,{hello,60}}
log: na ringo {received,{hello,60}}
log: na george {sending,{hello,1}}
log: na john {received,{hello,1}}
```

Example with more Jitter



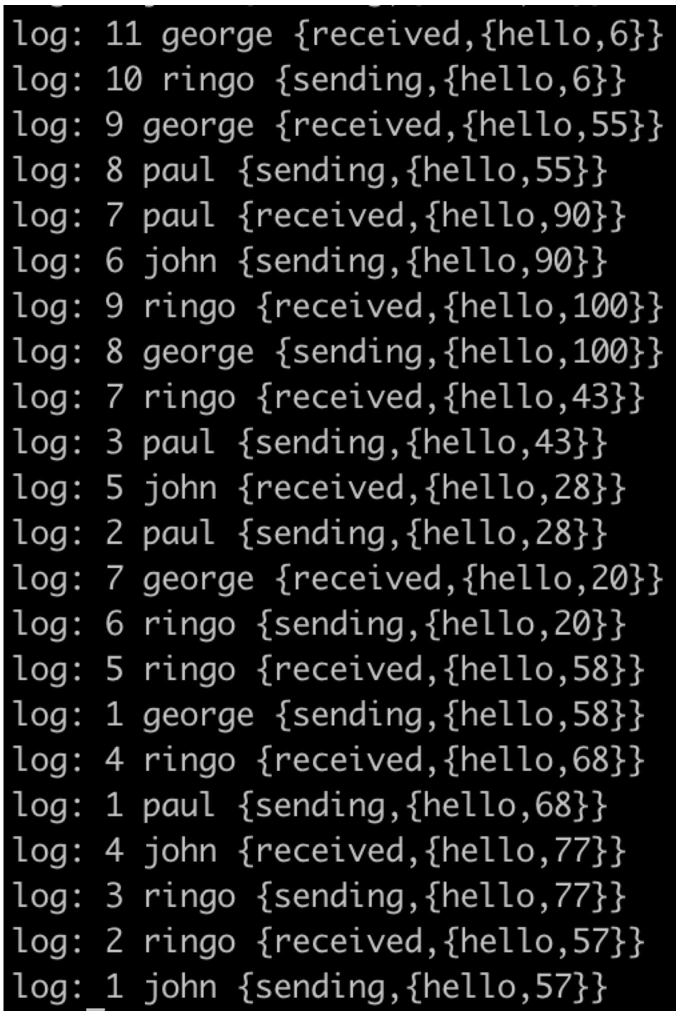
```
57> time_test:run(1000, 100).
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
log: na john {received,{hello,77}}
log: na ringo {sending,{hello,77}}
log: na ringo {received,{hello,68}}
log: na paul {sending,{hello,68}}
log: na george {received,{hello,20}}
log: na john {received,{hello,20}}
log: na ringo {sending,{hello,20}}
log: na paul {sending,{hello,20}}
log: na george {received,{hello,84}}
log: na john {sending,{hello,84}}
log: na george {received,{hello,16}}
log: na paul {sending,{hello,16}}
log: na paul {received,{hello,7}}
log: na george {received,{hello,97}}
log: na ringo {sending,{hello,97}}
```

You can determine if log messages are printed in the wrong order by checking the sequence of "sending" and "received" log messages for each process. The sending message of a specific process should of course appear before the received message.

No Jitter: When there’s no Jitter, so the sleep times are fixed, the order of the message sends is more predictable and the chance of messages being sent and logged in the wrong order is reduced in comparison to scenarios with higher Jitter

Higher Jitter: Higher Jitter increases the chance of messages being sent and logged in the wrong order because it leads to greater variability in the sleep times of worker processes

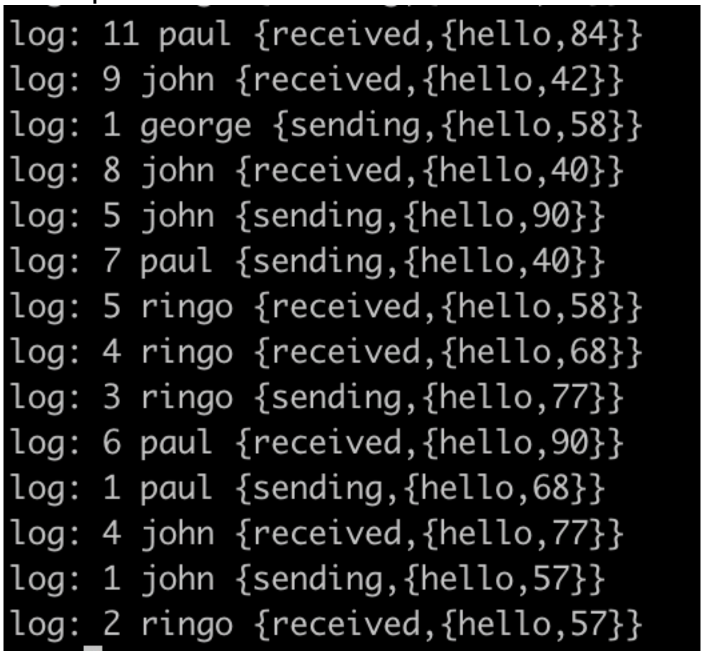
Example outcome with Lamport timestamps



```
log: 11 george {received,{hello,6}}
log: 10 ringo {sending,{hello,6}}
log: 9 george {received,{hello,55}}
log: 8 paul {sending,{hello,55}}
log: 7 paul {received,{hello,90}}
log: 6 john {sending,{hello,90}}
log: 9 ringo {received,{hello,100}}
log: 8 george {sending,{hello,100}}
log: 7 ringo {received,{hello,43}}
log: 3 paul {sending,{hello,43}}
log: 5 john {received,{hello,28}}
log: 2 paul {sending,{hello,28}}
log: 7 george {received,{hello,20}}
log: 6 ringo {sending,{hello,20}}
log: 5 ringo {received,{hello,58}}
log: 1 george {sending,{hello,58}}
log: 4 ringo {received,{hello,68}}
log: 1 paul {sending,{hello,68}}
log: 4 john {received,{hello,77}}
log: 3 ringo {sending,{hello,77}}
log: 2 ringo {received,{hello,57}}
log: 1 john {sending,{hello,57}}
```

The number before the process indicates the Lamport timestamp. So if we take the line at the bottom for example we can say that John sent a message “{hello, 57}” with a Lamport timestamp of 1.

Example with more Jitter


```
log: 11 paul {received,{hello,84}}
log: 9 john {received,{hello,42}}
log: 1 george {sending,{hello,58}}
log: 8 john {received,{hello,40}}
log: 5 john {sending,{hello,90}}
log: 7 paul {sending,{hello,40}}
log: 5 ringo {received,{hello,58}}
log: 4 ringo {received,{hello,68}}
log: 3 ringo {sending,{hello,77}}
log: 6 paul {received,{hello,90}}
log: 1 paul {sending,{hello,68}}
log: 4 john {received,{hello,77}}
log: 1 john {sending,{hello,57}}
log: 2 ringo {received,{hello,57}}
```

Write a report that describes your time module:

- Role of time module: module for managing Lamport timestamps within a distributed system. It provides functions to initialize and update Lamport clocks for individual nodes, compare timestamps for event ordering, and determine the causal relationships between events.
- zero/0: returns the initial Lamport timestamp, which is 0
- inc/1: takes a Lamport timestamp as an argument and increments it by one, representing the passage of time or the occurrence of an event
- merge/2: merges two Lamport timestamps by taking the maximum value. It ensures that when comparing timestamps from different nodes, the higher timestamp is retained. This is used to establish a partial order of events.
- leq/2: checks if one Lamport timestamp is less than or equal to another. This comparison is used to determine the causality relationship between events.
- clock/1: This function has a list of node names as argument. It creates an initial clock to keep track of the Lamport timestamps for each node. It initializes the clock with all timestamps set to zero.
- update/3: update the clock when a log message is received from a node at a given time. It calls update_clock/3 to handle the update operation and returns the updated clock.
- update_clock/3: helper function to update the clock for a specific node. It searches for the node in the existing clock. If the node is found, it merges the existing timestamp with the new timestamp and updates the clock. If the node is not found, it leaves the clock unchanged.
- safe/2: It has a Lamport timestamp and clock which contains timestamps for multiple nodes as argument. It checks if it's safe to log an event with the given timestamp. It ensures that the Lamport timestamp of the event is greater than or equal to the Lamport timestamps of all nodes in the clock, so that the event is causally related to all previous events.

How do you identify messages that are in the wrong order?

Messages are in the wrong order when the “received” event has a lower Lamport time, so timestamp, than the “sent” event from the same process.

What is always true, and what is sometimes true?

Always true: If a "received" event has a higher timestamp than the "sent" event from the same process, it's considered in the right order.

Sometimes true: If timestamps are equal, messages can still be in the right order.

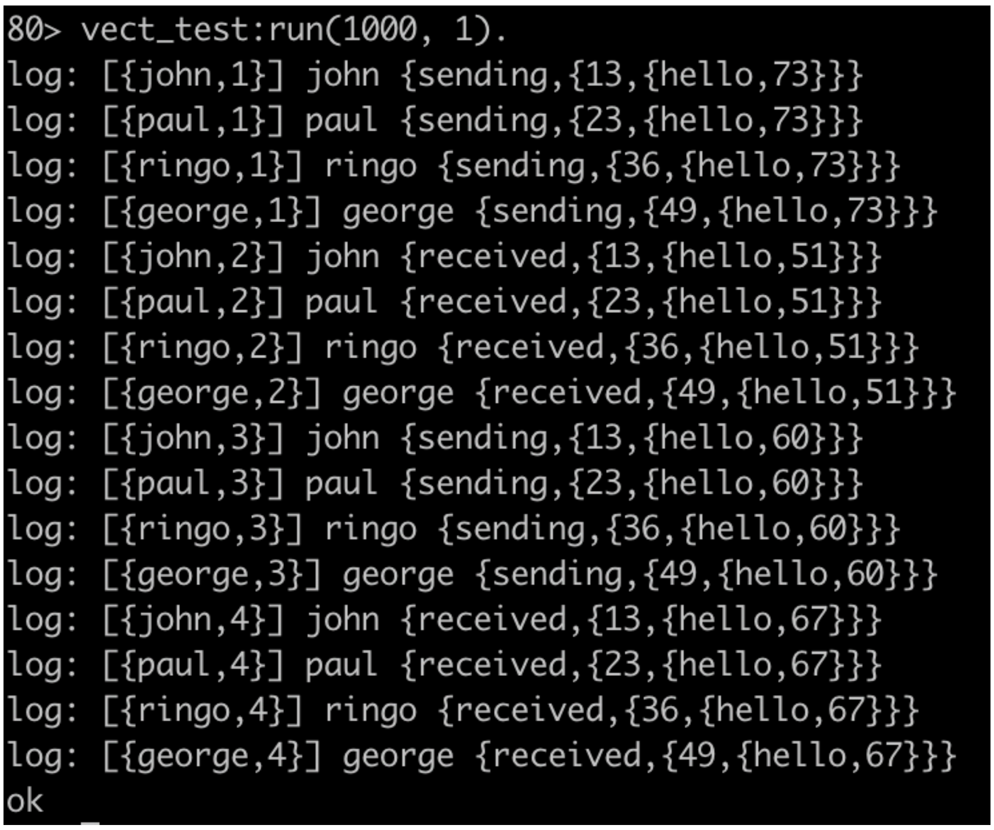
How do you play it safe?

You can play it safe by ensuring that when a worker receives message it adjusts its clock by choosing the later time between its own clock and the time mentioned in the message.
In addition to that the Lamport time, so timestamp, should be incremented before sending a message.

What is it that the final log tells us? Did events happen in the order presented by the log?

The final log tells us the order in which events were logged by the logger process based on Lamport timestamps.
However, what we get from the final log is not necessarily the real-time order in which events occurred in the system.
Messages can have delays and accordingly it could occur that a message sent later in real-time might arrive earlier than a previously sent message, so it results in a discrepancy between the Lamport timestamps and real-time order.
Another aspect are independent events because Lamport timestamps are primarily designed to capture causal relationships between events. Accordingly, if two events are independent Lamport timestamps do not provide a strict ordering between them. As a result they could appear in a wrong order.

Bonus



```
80> vect_test:run(1000, 1).
log: [{john,1}] john {sending,{13,{hello,73}}}
log: [{paul,1}] paul {sending,{23,{hello,73}}}
log: [{ringo,1}] ringo {sending,{36,{hello,73}}}
log: [{george,1}] george {sending,{49,{hello,73}}}
log: [{john,2}] john {received,{13,{hello,51}}}
log: [{paul,2}] paul {received,{23,{hello,51}}}
log: [{ringo,2}] ringo {received,{36,{hello,51}}}
log: [{george,2}] george {received,{49,{hello,51}}}
log: [{john,3}] john {sending,{13,{hello,60}}}
log: [{paul,3}] paul {sending,{23,{hello,60}}}
log: [{ringo,3}] ringo {sending,{36,{hello,60}}}
log: [{george,3}] george {sending,{49,{hello,60}}}
log: [{john,4}] john {received,{13,{hello,67}}}
log: [{paul,4}] paul {received,{23,{hello,67}}}
log: [{ringo,4}] ringo {received,{36,{hello,67}}}
log: [{george,4}] george {received,{49,{hello,67}}}
ok
```

Example explanation for outcome:
John (2) received a message from process 13: {13, {hello,51}}
John 2 indicates the state of the vector clock for process John at the time this log entry is generated. It says that the vector clock for John is currently at version 2. The version number is typically incremented when there is a local event or an update to the vector clock.
In this case the process named John receives a message which was sent by the process 13. {hello, 51} is the content of the message. Accordingly, the message has the type hello and the timestamp 51
When John receives this message, it updates its vector clock to incorporate the information from the received message. Specifically, it updates the entry for process 13 in its vector clock to the maximum of its current value and the logical timestamp of the received message (51 in this case).
This kind of mechanism helps track causality among events in a distributed system. If the logical timestamp of a received message is smaller than the local logical timestamp of the receiving process, it indicates that the message was sent in the past, and the vector clock is adjusted accordingly

Identified differences between the Lamport clock and the vector clock:

- Lamport clocks are based on the concept of "happened-before", stricter order in comparison to vector clocks
- Vector clocks are more flexible and allow for a more relaxed order of events, especially when events are not causally related
- Vector clocks are designed to handle concurrency more gracefully. Events that are causally independent can be ordered more flexibly in vector clocks.
- Lamport clocks may be more conservative in ordering events, potentially leading to a more serialized view of the system.
- When there are for example network delays Lamport clocks might result in more divergence between the clocks of different processes.
- In case network connection issues are fixed, vector clocks can more quickly and effectively synchronize information from different parts of the system compared to other clock mechanisms. This enables a faster and smoother alignment of events across the distributed system.

4 Conclusion

In summary, the assignment was very useful to better understand of how to work with logical time when sending data packages like for example messages. It was very interesting to discover which factors can influence the correct order of sent and received messages and to test it in different situations with different parameters. Furthermore, tt was interesting to observe the differences between the Lamport clock and the vector clock.