347: Distributed algorithms

Coursework 1

How to run

We have split our submission into 6 directories, each of them corresponding to 1 task. Each directory has a `Makefile` so to test our code, you can simply change to the directory corresponding to the desired task and run `make run`

Task 1 - Erlang Broadcast

```
{task1, start, 1000, 3000}
1: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
3: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
4: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
2: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
5: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
```

We were delighted to see that our output matches the expected output in the specification. Each process managed to send `Max_messages` and also to receive them.

```
{task1, start, 0, 3000}
3: {150592,161366} {150592,161338} {150592,150592} {150592,150592}
{150592,150598}
4: {150593,161366} {150593,161338} {150593,150592} {150593,150592}
{150593,150597}
5: {150598,161366} {150598,161338} {150598,150592} {150598,150592}
{150598,150597}
1: {161367,161367} {161367,161338} {161367,150592} {161367,150593}
{161367,150598}
2: {161338,161367} {161338,161338} {161338,150592} {161338,150592}
{161338,150598}
```

When we have infinite number of messages, vast majority of messages sent are also received. In our tests, there was a difference of at most one message between the number of sent messages and a number of received messages from the corresponding process. These messages are not lost, they are simply in the message queue of the receiving process after the `timeout` message, so they are ignored.

Task 2 - PL Broadcast

```
Max_messages = 100, Timeout = 1000
1: {100,100} {100,100} {100,100} {100,100} {100,100}
3: {100,100} {100,100} {100,100} {100,100} {100,100}
4: {100,100} {100,100} {100,100} {100,100} {100,100}
5: {100,100} {100,100} {100,100} {100,100} {100,100}
2: {100,100} {100,100} {100,100} {100,100} {100,100}
```

Similarly to task 1, we found nothing to surprising. When the number of messages is limited and the timeout is long enough, all the messages are sent and processed.

```
1: {13027,13012} {13027,13001} {13027,13004} {13027,13006} {13027,13009}
2: {13009,13005} {13009,12995} {13009,12997} {13009,12999} {13009,13002}
3: {13014,13006} {13014,12997} {13014,12999} {13014,13002} {13014,13003}
4: {13020,13010} {13020,13000} {13020,13002} {13020,13004} {13020,13007}
5: {13024,13010} {13024,13000} {13024,13003} {13024,13006} {13024,13007}
```

More interesting case is when the number of messages is unlimited. The number of receives no longer matches the corresponding number of sends. This is caused by adding another layer of abstraction.

The app has no idea what is happening bellow it, so it just keeps sending `pl_send` messages to Pl component and it gets immediately flooded. The `inter_pl` messages from other Pls are far in the queue and they get processed later. When the timeout message arrives to app, there is still a lot of `inter_pl` messages in message queues or Pls, but those get discarded.

This issue was becoming more and more pronounced in later tasks, when we were adding more layers of indirection. We were thus forced to find a way to ensure fairness when flooded by `pl_send`. We have outlined our solution in next part, where we think it is more relevant.

Task 3 - Best Effort Broadcast

```
Max_messages = 100, Timeout = 1000
1: {100,100} {100,100} {100,100} {100,100} {100,100}
3: {100,100} {100,100} {100,100} {100,100} {100,100}
4: {100,100} {100,100} {100,100} {100,100} {100,100}
5: {100,100} {100,100} {100,100} {100,100} {100,100}
2: {100,100} {100,100} {100,100} {100,100} {100,100}
```

As expected, there is equal number of send and received messages for each process and this number `Max_messages`.

```
Max_messages = 0, Timeout = 1000
1: {4567,193} {4567,192} {4567,192} {4567,191} {4567,192}
2: {4566,193} {4566,192} {4566,192} {4566,191} {4566,192}
3: {4565,193} {4565,192} {4565,192} {4565,191} {4565,192}
5: {4566,193} {4566,192} {4566,192} {4566,191} {4566,191}
4: {4565,193} {4565,192} {4565,192} {4565,191} {4565,191}
```

This is a more interesting case. At first, we were having problems with a very small number of messages received (in some cases it was 0) even tough the number of sent messages was going over the roof. We realized that this is likely caused by the fact that we are processing all sends and receives in FIFO order. By the time first message is received, each PL was flooded by send commands, so it would not get around to read it for a while. Moreover, there would be nothing stopping app module and beb module from flooding it by more broadcasts. We realized that the "send" messages are starving the receives.

Our original code in Pl looked a bit like this:

```
pl() ->
  receive
    {pl_send, _} -> send to another pl
    {pl_receive, _} -> send up to beb
  end.
```

Although it makes sense, it does not ensure that messages can be received in presence of a lot of sends. We have accomplished this behavior by refactoring the component to look a bit like this:

```
pl() ->
  receive
    {pl_send, _} ->
      send to another pl,
      deliver()
    after 0 -> deliver()
  end.
deliver() ->
  receive
    {inter_pl, _} ->
      send up to beb,
      pl()
  after 0 -> pl()
  end.
```

This ensured that if there is a message that needs to be delivered, it will happen even tough the pl is flooded by sends. Unfortunately, this has a performance penalty, as we need to scan the whole message queue in the worst case. We copied this pl implementation to Task 2 and used the same pattern in Beb and RB components as well.

Task 4 - Unreliable Message Sending

Limited messages

```
Max_messages = 100, Timeout = 1000, Rel = 100
1: {100,100} {100,100} {100,100} {100,100} {100,100}
3: {100,100} {100,100} {100,100} {100,100} {100,100}
4: {100,100} {100,100} {100,100} {100,100} {100,100}
5: {100,100} {100,100} {100,100} {100,100} {100,100}
2: {100,100} {100,100} {100,100} {100,100} {100,100}


Max_messages = 100, Timeout = 1000, Rel = 50
1: {100,54} {100,40} {100,47} {100,53} {100,46}
3: {100,52} {100,57} {100,51} {100,52} {100,54}
5: {100,52} {100,55} {100,49} {100,49} {100,43}
2: {100,44} {100,42} {100,56} {100,51} {100,54}
4: {100,43} {100,46} {100,60} {100,52} {100,51}

Max_messages = 100, Timeout = 1000, Rel = 0
4: {100,0} {100,0} {100,0} {100,0} {100,0}
1: {100,0} {100,0} {100,0} {100,0} {100,0}
2: {100,0} {100,0} {100,0} {100,0} {100,0}
5: {100,0} {100,0} {100,0} {100,0} {100,0}
3: {100,0} {100,0} {100,0} {100,0} {100,0}
```

When `Rel = 100`, the lossy links component has exactly the same behavior as the perfect links component, so the result is the same as we have seen before (all messages are delivered). In the opposite situation, when `Rel = 0`, the lossy links component just drops all the messages, so nothing gets delivered as expected. Finally, when `Rel = 50`, the number of received messages seems to be 50 on average and we believe it follows binomial distribution with 100 trials and probability p = `Rel`/100 = 0.5.

Unlimited messages

```
Max_messages = 0, Timeout = 1000, Rel = 100
4: {5723,236} {5723,260} {5723,260} {5723,257} {5723,238}
1: {5738,233} {5738,258} {5738,256} {5738,258} {5738,240}
3: {5751,244} {5751,262} {5751,260} {5751,258} {5751,246}
2: {5747,244} {5747,266} {5747,264} {5747,261} {5747,244}
5: {5756,235} {5756,258} {5756,257} {5756,257} {5756,239}

Max_messages = 0, Timeout = 1000, Rel = 50
3: {4757,192} {4757,192} {4757,192} {4757,192} {4757,191}
4: {4757,191} {4757,192} {4757,192} {4757,191} {4757,191}
2: {4757,192} {4757,192} {4757,192} {4757,192} {4757,191}
1: {4757,192} {4757,192} {4757,192} {4757,191} {4757,191}
5: {4757,191} {4757,192} {4757,192} {4757,191} {4757,191}

Max_messages = 0, Timeout = 1000, Rel = 0
1: {6090,0} {6090,0} {6090,0} {6090,0} {6090,0}
2: {6090,0} {6090,0} {6090,0} {6090,0} {6090,0}
3: {6090,0} {6090,0} {6090,0} {6090,0} {6090,0}
4: {6090,0} {6090,0} {6090,0} {6090,0} {6090,0}
5: {6085,0} {6085,0} {6085,0} {6085,0} {6085,0}
```

When `Rel = 0`, no messages are received as we would expect. More interesting fact arises from the comparison of `Rel = 100` and `Rel = 50`. We expected that the average number of receives is going to be smaller by 50% in case of `Rel = 50` but even after multiple experiments (to smoothen out the effect of cores being used by other processes) we did not really get this behavior.

We believe that the lower level components (`PL`,`BEB`) are "flooded" with broadcast and send commands by the time first message is received. Because of our decision to give equal opportunity to sends and receives, the system actually does not care what is the ratio of sends and receives in the message queue as long as there is enough commands to keep it busy. Therefore, even tough there are less (50%) received messages from other peers in the messsage queues of lower level components, almost the same number of them gets the opportunity to be delivered all the way to app component.

Task 5 - Faulty Process

After running with `Max_messages = 100`, `Timeout = 1000` and `process 3` termination after 12 ms we got the following result :

```
2: {100,100} {100,100} {100,13} {100,100} {100,100}
4: {100,100} {100,100} {100,12} {100,100} {100,100}
5: {100,100} {100,100} {100,12} {100,100} {100,100}
1: {100,100} {100,100} {100,13} {100,100} {100,100}
```

We can see that termination of `process 3` happened during its broadcast, and it managed to send message 13 to `process 1` and `process 2` but got cut off before it managed to send message 13 to `process 4` and `process 5`. This is expected behavior of best effort broadcast, as it has no guarantees that all correct eventually processes receive the same set of messages. We have chosen longer timeout

than suggested because otherwise the process always terminated before sending any message whatsover (but we found this to be machine specific).

After running with `Max_messages = 0`, `Timeout = 1000` and `process 3` termination after 30 ms we got the following result :

```
2: {5219,251} {5219,250} {5219,47} {5219,254} {5219,250}
5: {5219,251} {5219,250} {5219,46} {5219,254} {5219,250}
1: {5220,251} {5220,250} {5220,47} {5220,254} {5220,250}
4: {5219,251} {5219,251} {5219,47} {5219,255} {5219,250}
```

Similarly, `process 3` was terminated after sending message 47 to `process 1` but before sending the message to other processes. This issue will be resolved in task 6 after implementing reliable broadcast.

Task 6 – Eager Reliable Broadcast

We have decided to test the system with multiple sets of parameters to see how the system behaves under different conditions. We found the results of 3 cases particularly interesting:

- Limited messages and long timeout
- Limited messages and very short timeout
- Infinite messages

We did not find playing with `process 3` timeout particularly enlightening. With small values (3ms or less) it just finished without sending any message, otherwise the number of received messages increased approximately linearly with the timeout.

Limited messages and long timeout

After running with `Max_messages = 100` and `Timeout = 3000` we got the following result:

```
1: {100,100} {100,100} {100,2} {100,100} {100,100}
4: {100,100} {100,100} {100,2} {100,100} {100,100}
5: {100,100} {100,100} {100,2} {100,100} {100,100}
2: {100,100} {100,100} {100,2} {100,100} {100,100}
```

We can see that all processes finish sending `Max_messages` apart from `process 3`, who is killed earlier. On multiple runs, we see the number of messages received from `process 3` vary significantly, but it is always the same across all the correct processes, as we would expect from reliable broadcast.

Increasing the lossiness of the link, we see that the number of successful receives decreases slowly. On 80%, it seems that we still have more than 99% confidence that broadcasted messages will be seen by all hosts. On 50% reliability, hosts report on average 87 receives and on 10% reliability they report on average 15% receives (it should be noted that we stopped `process 3` from failing when testing the link reliability, to simplify our next computation.)

We tried to model this mathematically to find the actual probability of a message reaching a given destination from a given start point:

```
# P(N) is a probability that a link of length N fails

P(1) = (1 - rel)
P(N) = P (N - 1) * rel + ( 1 - rel)

Assuming 5 nodes, there are:
1 path from Start to Dest of Length 1
3 paths from Start to Dest of Length 2
6 paths from Start to Dest of Length 3
6 paths from Start to Dest of Length 4

The probability the message is delivered is then:

1 - P(1) * P(2) ^ 3 * P(3) ^ 6 * P(4) ^ 6

because we need all paths from start to destination fail in order for
destination node never to see a message.
```

Our experimental results seems to more or less follow this model. However, there is a known weakness that some of the longer paths were already invalidated by partial successful deliveries on shorter paths. We chose to ignore this, because we found it impossible to model.

Limited messages and short timeout

After running with `Max_messages = 100` and `Timeout = 300` we got the following result:

```
2: {100,38} {100,38} {100,2} {100,38} {100,38}
4: {100,38} {100,38} {100,1} {100,38} {100,38}
5: {100,38} {100,38} {100,1} {100,38} {100,38}
1: {100,38} {100,38} {100,2} {100,38} {100,38}
```

In this case, we can see that there is no agreement in the number of received messages from `process 3`. This is caused by the fact that we send the timeout signal so soon that `process 5` simply did not have enough time to process all rebroadcasts from other processes and therefore have not seen the second message yet (it is probably in deliver queue of its `PL`, `BEB` or `RB` component and have not reached `App` component responsible for counting). The agreement would be reached later on, if we did not timeout the system. This theory is also supported by observation that none of the processes had enough time to finish processing all the messages from correct peers.

Unlimited messages and long timeout

After running with `Max_messages = 0` and `Timeout = 3000` we got the following result:

```
1: {8013,288} {8013,288} {8013,23} {8013,290} {8013,289}
4: {8014,288} {8014,288} {8014,22} {8014,290} {8014,289}
2: {8012,288} {8012,288} {8012,23} {8012,290} {8012,289}
5: {8015,288} {8015,288} {8015,22} {8015,290} {8015,289}
```

In this final case, there is again no consensus reached about the number of the messages received from `process 3`. We attribute this to the fact that the low level components (`RB`,`BEB`,`PL`) are flooded with infinity messages from other peers, so even if the timeout is long, they did not have time to process all the rebroadcasts of messages from `process 3`. We can see even tough Eager Reliable Broadcast guarantees that all correct processes will *eventually* get all the messages, it is not always the case that it happens soon enough before we shut down the system.

When we set the timeout much higher (10s +), we always see that the system reaches consensus on number of messages received from `process 3`.