Distributed Algorithms: Coursework 1 Report

Daniel Yung (lty16) Tim Green (tpg16)

February 2019

1 Interleaving Implementation

In all of our broadcasting algorithms we ensure the messages are interleaved in each peer by rapidly switching between sending and listening. Initially, when a peer is started and all setup has been completed, it will do its first iteration of message sending. When this is complete, it will call `listen()` and begin listening for messages. Importantly, the timeout for the `receive` block in the listen function is set to 0. This means when the receive block is executed it will check its message queue for any of the messages listed, if none are waiting it will immediately timeout and go back to broadcasting again. It will continue in this fashion until either a `:timeout` message is received, or all intended messages have been broadcast. After finishing all the broadcasts the peer will recursively call `listen()` to finish processing all the messages in the queue. In both the listen and broadcast method, the peer will listen to a `:timeout` message sent by itself before starting the message broadcast. When it receives this it will stop processing immediately and print its status.

2 Broadcast 1

2.1 Implementation

Our implementation of Broadcast 1 peer to peer broadcast only has 2 modules.

Broadcast1. The Broadcast1 module spawns 5 Peers and passes in the Peer PID to the start function and sends the list of Peer PID to each of the Peers so they can communicate with each other. A broadcast instruction is then sent along with number of messages and max time out to each of the peers.

Peer. The Peer module starts by listening for the Peers list message ({ `:peers, peers` }) from the Broadcast1 module. Once it has received the Peers list it will wait for instructions from Broadcast1. In this case, a { `:broadcast, max_broadcasts, timeout` } instruction is sent from the network (Broadcast1 module). The Peer will firstly send a timeout message to itself with a delay which is specified in the broadcast message. Then it sends a message to all its Peers and calls the listen function where it will check its message queue to see if it has received any messages from other Peers. If it has received messages it will process 1 message and do another broadcast until the number of broadcast messages specified in the broadcast message has been reached. However if no messages are received it will go back to broadcasting to its peers. If all broadcast messages has been sent, peer will call listen and finish processing all the back log of messages. If at any point during this it receives a timeout message (which was sent earlier) it will stop processing and print it status.

2.2 Analysis.

Our interesting request we choose to do is { `:broadcast, 1000, 0` }. In this request we can see how many messages are sent before the peer actually receives the timeout.

```
Broadcast1 version 3
Peer 0:{94, 49},{94, 30},{94, 15},{94, 0},{94, 0}
Peer 2:{38, 38},{38, 0},{38, 0},{38, 0},{38, 0}
Peer 3:{33, 33},{33, 0},{33, 0},{33, 0},{33, 0}
Peer 1:{58, 48},{58, 10},{58, 0},{58, 0},{58, 0}
Peer 4:{94, 48},{94, 30},{94, 16},{94, 0},{94, 0}
```

Figure 1: The result from running the interesting request

One possible reason we suspect some messages are sent and received is because that while the Process.send_after is set to 0, the granularity of the system's clock is not as small as the elixir's process speed. Within the same time slice of the CPU clock, the elixir processes are able to send and receive some messages.

Another potential reason for the latency in the process timing out is the overhead associated with creating a new process, the line

```
Process.send_after(self(), { :timeout }, timeout)
```

will spawn a new asynchronous process, even with a timeout of zero [1]. Spawning a new asynchronous process takes time and resources. Given that this delay exists, the program will keep executing while the process is spawning and eventually encounter our `broadcast()` function which will begin sending messages. These messages will be added to the message queue of the current process before the new process has had time to spawn. This will further increase the time taken for `Peer` to process the `:timeout` message.

3 Broadcast 2

3.1 Implementation

Our implementation of Broadcast 2 Perfect Link broadcast has 4 modules.

Broadcast2. The Broadcast2 module is similar to Broadcast1 but is also responsible for binding all the PLs with each other to allow communication between them. When each Peer is spawned it will send its PID to the Broadcast2 module using a { `:bind_bc_pl` } message. Once Broadcast2 has received messages from all Peers it will broadcast a global list of all PLs to all other PLs (including themselves). Finally the broadcast instruction ({ `:broadcast, msg_num, timeout` }) is sent to all Peers.

Peer. The Peer module is responsible for spawning all the components within itself. In Broadcast 2 it will spawn the Com module and the PL module. It is also responsible for passing the broadcast instruction from the network into the Com component.

Com. The Com module is the module that keeps track of the `sent` and `received` array and is responsible for printing the results to console. Com first binds itself to its respective PL component through a { `:pl_com_bind` } message. It will then listen for instructions from Peer which, in this exercise, is just to broadcast. Com is similar to Peer in Broadcast1, with the difference that the messages are sent via the PL module using { `:pl_send` } instead of sending directly to the Peers. The messages sent to PL contains only the recipient index (index within the global PL list). The Com module also receives messages from PL which contains the sender's index so it can update the received list.

PL. The PL module is responsible for the direct message transferring from 1 Peer to another. PL will receive a send request message from its Com { `:pl_send` } with the recipient index where PL will extract the corresponding PID value in the PL list and send a { `:pl_deliver` } message directly to the recipient's PL. Upon receiving a { `:pl_deliver` } from another Peer's PL, it will pass on a { `:received` } message to it's respective Com module so that it can account for the messages received.

3.2 Analysis

The results from our { `:broadcast, 10_000_000, 3000` } request shows that there are certain peers that sends a lot more than other peers. Peer one sends 428445 messages unlike Peer 3 who only sent 1198 messages.

```
mix run --no-halt -e Broadcast2.main 2 5
Broadcast2 version 2
Peer 1:{428445, 2021},{428445, 70685},{428445, 1821},{428445, 1035},{428445, 1453}
Peer 3:{1198, 167},{1198, 235},{1198, 162},{1198, 205},{1198, 205}
Peer 0:{2151, 242},{2151, 299},{2151, 237},{2151, 518},{2151, 294}
Peer 4:{1622, 198},{1622, 265},{1622, 229},{1622, 515},{1622, 249}
Peer 2:{1985, 333},{1985, 431},{1985, 269},{1985, 531},{1985, 368}
```

Figure 2: The result from running the timeout request

In our interesting request we reduce the timeout by half to 1500. The disparity seen in the previous request still exists however it is less prevalent. Peer 1 has the highest number of sent with 176756 and the lowest number of sent being Peer 3 with 747.

As we half the timeout, we would expect the messages sent and received to be approximately halved. In our result we can see that the total broadcast messages sent by each peer in the case where the timeout is set to be 1.5 seconds is 181,251. When the timeout is set to be 3 seconds it is 435,401. This shows a decrease by a factor of approximately 2.4 when the timeout is halved. We would expect a decrease by a factor of approximately 2. The extra decrease can be attributed to the inherent non-deterministic nature of asynchronous programs. The results can vary drastically each time the program is run.

```
mix run --no-halt -e Broadcast2.main 3 5
Compiling 1 file (.ex)
Broadcast2 version 3
Peer 3:{747, 73},{747, 307},{747, 191},{747, 69},{747, 63}
Peer 4:{762, 73},{762, 307},{762, 202},{762, 69},{762, 67}
Peer 1:{176756, 772},{176756, 14158},{176756, 1805},{176756, 552},{176756, 536}
Peer 2:{1977, 167},{1977, 893},{1977, 461},{1977, 195},{1977, 217}
Peer 0:{1009, 99},{1009, 416},{1009, 272},{1009, 89},{1009, 89}
```

Figure 3: The result from running the interesting request

4 Broadcast 3

4.1 Implementation

Our implementation of Broadcast 3 Best Effort Broadcast has 5 modules.

The Broadcast 3 module does not change from broadcast 2.

The Peer module now also spawns the Beb module but otherwise unchanged from broadcast 2.

The Com module now no longer send messages to PL but sends { `:beb_broadcast`} messages to it's Beb Module.

The Beb module is responsible for sending out the broadcast messages to PL when it receives a { `:beb_broadcast`} message from it's Com module. It also passes the { `:pl_deliver`} messages back up to the Com module for messages counting.

The PL module do not change from Broadcast 2.

4.2 Analysis

Here is the result when we send the peer the { `:broadcast, 10_000_000, 3000` } request.

```
Broadcast3 version 2
Peer 2:{967114, 90},{967114, 82},{967114, 0},{967114, 82},{967114, 78}
Peer 3:{232051, 18158},{232051, 18481},{232051, 13239},{232051, 13190},{232051, 11492}
Peer 4:{687725, 590},{687725, 541},{687725, 473},{687725, 533},{687725, 533}
Peer 0:{403652, 4},{403652, 6},{403652, 0},{403652, 0},{403652, 0}
Peer 1:{214858, 29868},{214858, 26502},{214858, 21060},{214858, 20665},{214858, 17271}
```

Figure 4: The result from running the interesting request

Our Interesting request for Best Effort Broadcast, we decided to double the timeout of the second request, sending { `:broadcast, 10_000_000, 6000` } request to each peer.

```
Broadcast3 version 3
 Peer 4:{1048047, 47},{1048047, 47},{1048047, 45},{1048047, 44},{1048047, 47}
Peer 2:{550809, 4230},{550809, 3261},{550809, 6144},{550809, 6532},{550809, 2900}
Peer 1:{639894, 48},{639894, 47},{639894, 0},{639894, 0},{639894, 0}
Peer 3:{575370, 54216},{575370, 37734},{575370, 47658},{575370, 53113},{575370, 42363}
Peer 0:{860176, 48},{860176, 18},{860176, 0},{860176, 0},{860176, 0}
```

Figure 5: The result from running the interesting request

Similar to Broadcast 2, when we double our timeout we would expect double the messages sent. However this isn't the case in Best Effort Broadcast. When the timeout is 6000 milliseconds the total amount of broadcast messages sent is 3,674,296, while the total amount of broadcast message sent when timeout is 3000 milliseconds is 2,505,400.

One explanation for this result is that the extra BEB module in between the Com and PL module which increases the latency per message. At such large volume, the additional overhead of receiving and forwarding a message will compound and increase the total time taken for the message to propagate through the system. Furthermore, the BEB module also handles the message receiving from PL to Com module. This means while Com tells BEB to broadcast, it could be handling a { `:pl_deliver` } message instead.

5 Broadcast 4

5.1 Implementation

Our implementation of Broadcast 4 Unreliable Message Sending Broadcast has 5 modules.

Most of the modules are unchanged except for the PL modules from Broadcast 4 has become LPL modules.

Broadcast 4 now takes a reliability parameter in the main function.

The LPL modules now take a reliability parameter from its peer so to simulate unreliable broadcasting. When a { `:pl_send` } message is received, LPL will randomly generate an integer in range from 1 - 100, if the number is smaller than the reliability (range from 0 - 100) then the message will be sent. Even though the message wasn't sent by PL, the Com module does not know this, it still count the message as sent this is to simulate unreliable message sending

5.2 Analysis

Follow are the results for 100% , 50% and 0% reliability when sending { `:broadcast, 1000, 3000` } request.

```
mix run --no-halt -e Broadcast4.main 100 5
Compiling 1 file (.ex)
Peer 4:{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000}
Peer 0:{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000}
Peer 3:{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000}
Peer 2:{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000}
Peer 1:{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000},{1000, 1000}
```

Figure 6: Broadcast4 running with 100% reliability

```
mix run --no-halt -e Broadcast4.main 50 5
Compiling 1 file (.ex)
Peer 0:{1000, 498},{1000, 488},{1000, 477},{1000, 504},{1000, 467}
Peer 3:{1000, 507},{1000, 501},{1000, 491},{1000, 476},{1000, 500}
Peer 1:{1000, 525},{1000, 481},{1000, 491},{1000, 503},{1000, 485}
Peer 4:{1000, 491},{1000, 526},{1000, 515},{1000, 502},{1000, 493}
Peer 2:{1000, 482},{1000, 520},{1000, 488},{1000, 501},{1000, 503}
```

Figure 7: Broadcast4 running with 50% reliability

```
mix run --no-halt -e Broadcast4.main 0 5
Peer 0:{1000, 0},{1000, 0},{1000, 0},{1000, 0},{1000, 0}
Peer 4:{1000, 0},{1000, 0},{1000, 0},{1000, 0},{1000, 0}
Peer 3:{1000, 0},{1000, 0},{1000, 0},{1000, 0},{1000, 0}
Peer 2:{1000, 0},{1000, 0},{1000, 0},{1000, 0},{1000, 0}
Peer 1:{1000, 0},{1000, 0},{1000, 0},{1000, 0},{1000, 0}
```

Figure 8: Broadcast4 running with 0% reliability

The results shown in Figures 6, 7, and 8 are to be expected. Because the number of send messages are counted in the Com module, we see 100% success in messages sends. However, the decision to send a message based on reliability is taken in the LPL module. This means when a { `:pl_send` } message is received in LPL it may not actually be sent to the destination LPL. As such, if the message is not sent due to the reliability the receiving LPL will not get the message and the receive count will not be incremented.

Here is the result for our interesting request that sends the same request but with a 0 timeout.

```
mix run --no-halt -e Broadcast4.main 100 5
Peer 1:{105, 0},{105, 0},{105, 0},{105, 0},{105, 0}
Peer 0:{736, 0},{736, 0},{736, 0},{736, 0},{736, 0}
Peer 3:{1000, 0},{1000, 0},{1000, 0},{1000, 0},{1000, 0}
Peer 2:{630, 0},{630, 0},{630, 0},{630, 0},{630, 0}
Peer 4:{1000, 0},{1000, 0},{1000, 0},{1000, 0},{1000, 0}
```

Figure 9: Broadcast4 running with 0 seconds timeout with 100% reliability

Similar to the results for previous broadcasts, we see that no messages are received. However, in this case, this is due to the timeout. Making the timeout zero means that some messages are able to send (we would expect all to success with 100% reliability) but mid-way through the process is terminated by the timeout. As a result no messages are received, likely because the path a message must take to be counted as received goes through 5 different communications between modules ($Com \rightarrow Beb \rightarrow LPL \rightarrow Beb \rightarrow Com$). A timeout message will reach the Com module much quicker, even with the associated overhead of spawning a new process for the timeout message.

6 Broadcast 5

6.1 Implementation

Our implementation of Broadcast 5 Faulty Process Broadcast has 5 modules. Most modules are unchanged from Broadcast 4 except for the Peer and Com modules. When Peer 3 is initialised, it will send a kill signal to its corresponding Com module after 5 milliseconds. When this happens Com will call `Process.end()` on itself to simulate a faulty process.

6.2 Analysis

Our interesting request send for Broadcast 5 is to send each peer { `:broadcast, 5000, 3000` } with 1% reliability broadcast request. The following figure shows the result of this request.

```
KILLED:Peer 3:{105, 0},{105, 0},{105, 0},{105, 0},{105, 0}
Peer 0:{5000, 44},{5000, 46},{5000, 55},{5000, 1},{5000, 49}
Peer 1:{5000, 49},{5000, 49},{5000, 47},{5000, 0},{5000, 60}
Peer 2:{5000, 51},{5000, 32},{5000, 43},{5000, 2},{5000, 52}
Peer 4:{5000, 45},{5000, 51},{5000, 51},{5000, 1},{5000, 38}
```

Figure 10: Broadcast5 running with 5 seconds timeout with 100% reliability

As you can see Peer 3 is killed after 5 milliseconds and within that period of time, Peer 3 manage to send 105 broadcast messages to all of its peers. In this situation Peer 1 doesn't know that Peer 3 exists or has sent any messages since it received 0 messages from Peer 3 while other Peers have received at least 1 message. Peer now have inconsistent information about each other. If these were heart beat messages, Peer 1 would conclude that Peer 3 does not exist while the others might just believe that Peer 3 has connections issues. A possible solution to this issue in a real world scenario would be to use the architecture in Broadcast6. In Broadcast6 when a message is received it is re-broadcast for all other Peers to see. This would remove the inconsistent information as other Peers would inform Peer 1 that Peer 3 exists.

7 Broadcast 6

7.1 Implementation

Our implementation of Broadcast 6 Eager Reliable Broadcast has 6 modules. Up to this point all messages sent between PL do not contain any unique ID or Sequence number. However, in this broadcast algorithm our messages contain the sender's Com PID and a sequence number { `m_sender_index, seq_num` }. We added this feature because in each Erb Module, it must be able to differentiate if a package have been received before in order to avoid duplication when rebroadcasting messages. If a peer died after broadcasting a message, that message will

be rebroadcast on behalf of other peers that received this message. Most modules are similar to Broadcast5, only changed to accomodate for the Erb module.

The Com module now sends the Erb module a unique sequence number and its own PID in a { `:rb_broadcast` } message. The Erb module receives the { `:rb_broadcast` } signal containing the PID of Com and a sequence number and wraps it in a message then it is sent to Beb in a { `:beb_broadcast` } message. The Erb module first initialises with an empty array for it's delivered messages. When it receives a { `:beb_deliver` } message, it will first check if the message is in it's delivered list, if it is not delivered array erb will send a { `:beb_broadcast` } message with this new message and add it into the delivered message array to avoid duplication of re-broadcasting and a { `:rb_delivered` } with the original recipient index of the message will be sent upstream to the Com module. In the implementation, broadcast sent by dead or faulty Peers will still be received by other alive Peers However if the message has already been re-broadcast, it will simply recurse and call listen again as we have heard this message already.

The Beb Module now also pass along the unique message from Erb to PL in a { `:pl_send` } message.

The LPL module now also pass along the unique message to the other LPL.

7.2 Analysis

```
Broadcast6 reliability: 1
Peer 3:{315, 0},{315, 0},{315, 0},{315, 0},{315, 0}
Peer 0:{5000, 59},{5000, 40},{5000, 46},{5000, 10},{5000, 38}
Peer 4:{5000, 67},{5000, 49},{5000, 48},{5000, 4},{5000, 46}
Peer 2:{5000, 56},{5000, 55},{5000, 46},{5000, 3},{5000, 38}
Peer 1:{5000, 50},{5000, 55},{5000, 38},{5000, 2},{5000, 54}
```

Figure 11: Broadcast6 sends 5000 messages 1% reliability and Peer 3 dying after 5 milliseconds

This is the exact condition as Broadcast5, but in this algorithm, every Peer now has consistent information with everyone in the network, due to the Eager Reliable Broadcast. If just 1 Peer have received a message from Peer 3, every peer will receive the broadcast message since every peer re-broadcast messages received.

References

[1] Elixir Documentation, *Process*, https://hexdocs.pm/elixir/Process.html#send_after/3