Distributed Algorithm 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.

When we run our solution in a single elixir node. Our results for { `:broadcast, 1000, 3000`} with 5 peers is as expected. All 5 peers send and receive all messages well within the 3 second timeout. However when running this solution with { `:broadcast, 10_000_000, 3000`}, the timeout is too short for all messages to be sent and received, what we end up seeing is a almost 5:1 sent to received messages per peer. This is due to our implementation of broadcast where we only process 1 message per broadcast. In an ideal situation where the timeout is long enough for

each peer to send all its broadcast messages, so it can have enough time after to process all the remaining messages received.

Our interesting request we choose to do is a to send { `: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 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, in our results we can see that elixir processes are able to send and receive some messages mainly for the first few peers that were spawned first and have a head start on other peers.

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 which 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 broadcast 1, however it is also responsible for binding all the PL with each other. It spawns all the Peer modules like in broadcast 1. After all the Peers have been spawned it will wait until all the PL has sent all their respective PID and index to the network in a { `:bind_pl` } message. When all the PL PID has been received the PL list will be sent to all the PL so they all can communicate with each other. Finally the broadcast instruction is sent to all the 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 will be the module that keeps track of the sent and received array and is responsible for printing the results to console. Com first bind itself with the PL component through a { `:pl_bind` } message then it will listen for instructions from Peer which in this exercise is just to broadcast. Com is similar to Peer in broadcast 1, however the messages are sent to PL { `:pl_send` } instead of sending directly to the peers. The messages sent to PL contains only the recipient ID in order for PL to know which peer's PL to send to. 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

For the standard { `:broadcast, 1000, 3000` } request we get what you would expect to receive. All peers receive and send 1000 messages.

However 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

This disparity between is also shown in our interesting request which halfs the timeout from the 10 million send request { `:broadcast, 10_000_000, 1500` }. the. The desparity is less however,peer 1 has the highest number of sent with 176756 and the lowest number of sent being peer 3 with 747.

We can see that our ex

```
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

4.2.1 Running solution locally

4.2.2 Running on Docker locally

5 Broadcast 4

5.1 Implementation

Our implementation of Broadcast 4 Lossy Perfect Link 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.

5.2 Analysis

..... Broadcasting 1000 messages at 100% 50% 0% Broadcasting 10,000,000 messages at 100% 50% 0%

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 the Com module after 5 milli-seconds where Com will call `Process.end()` on itself to simulate a faulty process.

6.2 Analysis

7 Broadcast 6

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 {`message_pid, 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.

Most modules are changed slightly from Broadcast 5.

The Com module now send 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. Erb Module first initials with an empty array for it's delivered messages. When the Erb module 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`} will be sent upstream to the Com module. 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.

7.1 Implementation

7.2 Analysis

References

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