Imperial College London

Distributed Algorithms - 347

Imperial College London

Department of Computing

CW Report

Authors:
Panayiotis Panayiotou - pp3414
Adrian Catana - ac7815

February 7, 2018

README.md is available for each of the systems to explain how to run it and how to pass arguments to it

Chapter 1

System 1

For the first system we use the peer process ids to distinguish the different processes. This is convenient as peers know where to send the messages too. We keep a map of sent and received messages in the peer and we interleave messages the following way.

For every broadcast, we iterate through the peers and firstly check if we got a timeout message. then check if we have any message to receive in our mailbox and then send the message to that peer. We did a nested receive when checking for timeout and for any messages in the mailbox so that we could give artificial priority to the timeout message because of the way "receive" is handled in elixir if we had timeout and incoming messages on the same receive then a timeout coming to a mailbox with a lot of messages already would have to wait until it is served - therefore not really acting correctly as a timeout. This way we enforce more strictly the timeout. This is also used when we have broadcast all messages and we call a method to wait for a timeout or an incoming message. Because we again wanted to enforce priority on any timeout message we used nested "receive"s with 0 timeout to check first for the timeout message and then for any incoming message.

Something we should mention is that we initially created a separate module that is a timer and send an :timeout message when the timeout given runs out. This proved to be unreliable due to how the Elixir node might schedule the different processes. We observed weird behavior related to this by setting the timeout to 0 and observing that sometimes we still send and receive quite a few messages (hundreds) before exiting.

The tests for this system were run on the following machine:

```
product: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
```

Local

The following test behaves as expected, since the number of broadcasts is low, fitting within the given time. Actually, by trying out different values, we discovered that, depending on scheduling, for this low number of broadcasts, our system can send and receive all the messages in less than 300 ms, so 1 second is more than enough!
Previously, we though it could be less than 20 ms. The reason for this is that we were using a self made Timeout module, a process spawned by the peer, that would send a { :timeout } message to the peer after timeout seconds. This is, however, wrong. This process might get unfair scheduling, thus the counting would be done wrongly. Instead, we are using :timer.send_after(timeout, { :timeout }), which behaves as expected.
We discovered this issue by trying to set timeout to 0. We had this strange behavior: we actually had messages that were broadcast! Thus, we add this test below, considering it helped us a lot to find out this issue!

```
Test: elixir processes, 5 peers, max 1000 broadcasts, after 3000 ms timeout
mix run --no-halt -e System1.main 5 1000 3000
#PID100 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
#PID103 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
#PID99  {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
#PID101 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
```

```
#PID102 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}

Test: elixir processes, 5 peers, max 1000 broadcasts, after 0 ms timeout
mix run --no-halt -e System1.main 5 1000 0
#PID101 {0, 0} {0, 0} {0, 0} {0, 0} {0, 0}
#PID99  {0, 0} {0, 0} {0, 0} {0, 0} {0, 0}
#PID102 {0, 0} {0, 0} {0, 0} {0, 0} {0, 0}
#PID100 {0, 0} {0, 0} {0, 0} {0, 0} {0, 0}
#PID103 {0, 0} {0, 0} {0, 0} {0, 0} {0, 0}
```

Testing something more extreme, we allow a maximum of 1 million broadcast messages with a timeout of 6 seconds. Now, there are 2 points to consider: the number of sent messages is at about a magnitude greater than the number of received messages and there was not enough time to do all the broadcasts. This happens due to our implementation design: once the set up is done (the peer receives all the messages), we keep on broadcasting messages. At the same time, in the same place, we check if there is any timeout/message received in the mailbox, but we do not wait for them. This increases the interleaving and ensures that we do not wait to send all the messages before we receive from other peers. Furthermore, we check for the case when we broadcast max_broadcast times and we are yet to receive messages (we only terminate our program on timeout). These principles are extensively used in the later systems.

```
Test: elixir processes, 5 peers, max 1000000 broadcasts, after 6000 ms timeout
mix run --no-halt -e System1.main 5 1000000 6000
#PID<0.99.0> {66278, 66278} {66278, 385} {66277, 371} {66277, 16911} {66277, 439}
#PID<0.100.0> {385, 858} {385, 85} {385, 47} {385, 758} {385, 70}
#PID<0.103.0> {439, 1039} {439, 88} {439, 50} {439, 941} {438, 76}
#PID<0.102.0> {16911, 20303} {16911, 128} {16911, 90} {16910, 16389} {16910, 156}
#PID<0.101.0> {371, 877} {371, 85} {371, 47} {371, 774} {370, 71}
```

Here is an example when the time is enough for some processes to sen all their messages, to receive all the messages, while for some other processes the time is not enough. This is mainly due to unfair scheduling and congested network. Basically, we have the following scenario, in concordance with our implementation: a message gets to send many messages to the others. Then, the others, before sending messages, check and find out that they received the message from the first process. Thus, they need to to some more computations before broadcasting, precious time when the first process will "race" faster and get to send more messages before needing to check its own.

```
Test: elixir processes, 5 peers, max 10000 broadcasts, after 2000 ms timeout
mix run --no-halt -e System1.main 5 10000 2000
#PID<0.99.0> {437, 46} {437, 1292} {437, 17} {437, 708} {436, 12}
#PID<0.102.0> {10000, 437} {10000, 10000} {10000, 732} {10000, 10000} {10000, 456}
#PID<0.101.0> {733, 56} {733, 2042} {732, 37} {732, 1505} {732, 22}
#PID<0.100.0> {10000, 437} {10000, 10000} {10000, 733} {10000, 10000} {10000, 457}
#PID<0.103.0> {457, 47} {457, 1379} {456, 19} {456, 824} {456, 13}
```

We run another test with just one peer to test this corner case

```
Test: elixir processes, 1 peer, max 1000 broadcasts, after 2000 ms timeout
mix run --no-halt -e System1.main 1 1000 3000
#PID<0.133.0> {1000, 1000}
```

Docker

For this section, we simply try the testcases from local on a Docker network. There is nothing particularly special about this test cases, since, again, 3 seconds is more than enough time to broadcast and receive everything. However, by testing, we have found that for 1000 broadcasts our system has problem to send/receive everything when we set the timeout to less than 100 ms. This is slower than the local network, somehow expected. Network transfers between two containers take longer compared to a setup with the client code running natively outside of Docker on the same host. See here for reference and benchmarks.

```
MAIN=System1.main_net PEERS=5 MAX_BROADCASTS=1000 TIMEOUT=3000 docker-compose -p da347 up
Attaching to container5, container1, container2, container4, container3, container0
```

```
container0 |[#PID<0.109.0> {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
container0 |[#PID<0.109.0> {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
container0 |[#PID<0.109.0> {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
container0 |[#PID<0.109.0> {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
container0 |[#PID<0.109.0> {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
```

Again, using the extreme example (2nd from the local section) we can see that the gap is even bigger between local and Docker networks, for the same reasons as above.

```
MAIN=System1.main_net PEERS=5 MAX_BROADCASTS=1000000 TIMEOUT=6000 docker-compose -p da347 up
Attaching to container3, container2, container5, container1, container4, container0
PID109 {42227, 10905} {42227, 308} {42227, 40411} {42227, 238} {42226, 319}
PID109 {665, 2040} {664, 8} {664, 1137} {664, 22} {664, 112}
PID109 {14374, 13687} {14373, 227} {14373, 3526} {14373, 130} {14373, 209}
PID109 {837, 2406} {836, 156} {836, 1445} {836, 32} {836, 140}
PID109 {853, 2672} {853, 7} {853, 1451} {853, 21} {853, 112}
```

Chapter 2

System 2

When adapting the system1 to system2 we chose to use the process ids of the PLs as the process ids as this way we could easily address processes from other processes (we knew exactly where to send a message to). We though this could introduce future problems (and it did in the extension of system 6) if we ever used more than 1 PL per process. The app component holds all the logic and sends/receives messages using the PL component. All the PL component is doing is forwarding messages to some other processes. We use the same mechanisms for timeouts as for system 1 The peer in this system acts as a constructor for the other classes and it waits for a timeout message when the app exits to also kill the PL process

Local

The test from below behaves as expected. Also, testing with timeout 0 gives same result as in system 1.

```
mix run --no-halt -e System2.main 5 1000 3000
PID106 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID110 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID104 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID108 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID112 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
```

Overall, testcase 2 is worse in terms of overall sent messages, but that is expected, since we are adding another layer of transmission.

```
mix run --no-halt -e System2.main 5 1000000 6000
PID117 {38383, 1722} {38383, 6332} {38382, 4588} {38382, 2472} {38382, 7814}
PID115 {6670, 551} {6670, 5257} {6670, 189} {6669, 639} {6669, 5726}
PID121 {8267, 548} {8267, 5254} {8267, 185} {8267, 637} {8266, 5316}
PID113 {2429, 544} {2429, 5251} {2429, 179} {2429, 636} {2429, 4640}
PID119 {2847, 616} {2847, 5269} {2847, 754} {2847, 646} {2846, 6104}
```

Docker

The first Docker test behaves as expected.

```
MAIN=System2.main_net PEERS=5 MAX_BROADCASTS=1000 TIMEOUT=3000 docker-compose -p da347 up
Attaching to container3, container1, container2, container5, container4, container0
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}

MAIN=System2.main_net PEERS=5 MAX_BROADCASTS=1000000 TIMEOUT=6000 docker-compose -p da347 up
Attaching to container5, container2, container4, container3, container1, container0
PID110 {12477, 1334} {12477, 1601} {12477, 1} {12477, 1} {12477, 0}
PID110 {15203, 1136} {15203, 1517} {15203, 0} {15203, 0} {15203, 0}
```

```
PID110 {7658, 2447} {7658, 1} {7658, 1} {7658, 1} {7657, 1}
PID110 {18819, 1861} {18819, 2024} {18819, 1} {18819, 1} {18819, 1}
PID110 {13450, 1066} {13450, 1714} {13449, 85} {13449, 1} {13449, 0}
```

Chapter 3

System 3

Again the Peer orchestrates the creation of the app, beb and pl and waits for a timeout message when app exits to also kill beb and pl processes. The app sends a broadcast message to the beb and then increments the counts for sent messages because as far as it's concerned the messages were sent. (faulty links or failure to deliver the messages might be possible but the app only cares how many messages it sent)

Local

```
mix run --no-halt -e System3.main 5 1000 3000
PID104 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID107 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID110 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID116 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
PID113 {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}

mix run --no-halt -e System3.main 5 10000000 9000
PID154 {7486, 4250} {7486, 7021} {7486, 5537} {7485, 5432} {7485, 3738}
PID163 {4279, 3234} {4279, 5895} {4279, 4749} {4279, 4431} {4279, 2871}
PID151 {4813, 3341} {4813, 6123} {4813, 4892} {4813, 4562} {4813, 2955}
PID160 {6052, 4353} {6051, 7133} {6051, 5610} {6051, 5541} {6051, 3835}
PID157 {5886, 4041} {5886, 6811} {5885, 5387} {5885, 5204} {5885, 3550}
```

Docker

```
MAIN=System3.main_net PEERS=5 MAX_BROADCASTS=1000 TIMEOUT=3000 docker-compose -p da347 up
Attaching to container3, container1, container2, container5, container4, container0
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}
{1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000} {1000, 1000}

MAX_BROADCASTS=10000000 TIMEOUT=9000 docker-compose -p da347 up
Attaching to container5, container2, container1, container4, container3, container0
PID110 {5010, 6193} {5010, 4735} {5010, 4986} {5010, 4644} {5010, 4633}
PID110 {6771, 6672} {6770, 5101} {6770, 5315} {6770, 4929} {6770, 4990}
PID110 {5193, 6206} {5192, 4750} {5192, 4999} {5192, 4655} {5192, 4660}
PID110 {5395, 6492} {5395, 4944} {5395, 5179} {5395, 4823} {5395, 4873}
PID110 {5053, 6224} {5053, 4758} {5052, 5007} {5052, 4660} {5052, 4673}
```

Chapter 4

System 4

Local

For unreliable message passing we just added a reliability argument to the PLs. On each broadcast we generate a random number from 1 to 100 (inclusive) and if the number is <= to the reliability we broadcast the message. This gives the appropriate probabilities of transmission. e.g. for reliability of 0 no number will be less than that (0%), for reliability of 1 only 1 number out of 100 will be less that that (1%) etc One can pass an argument for the LPL reliability through the makefile by setting the variable LPL_RELIABILITY = 100 LPL_RELIABILITY = 50 LPL_RELIABILITY = 0

```
mix run --no-halt -e System4.main 5 1000 3000 50
PID107 {1000, 511} {1000, 502} {1000, 507} {1000, 514} {1000, 497}
PID116 {1000, 507} {1000, 516} {1000, 504} {1000, 496} {1000, 515}
PID104 {1000, 523} {1000, 504} {1000, 516} {1000, 508} {1000, 517}
PID110 {1000, 507} {1000, 492} {1000, 511} {1000, 504} {1000, 502}
PID113 {1000, 524} {1000, 487} {1000, 531} {1000, 494} {1000, 477}

mix run --no-halt -e System4.main 5 1000 3000 0
PID107 {1000, 0} {1000, 0} {1000, 0} {1000, 0} {1000, 0}
PID116 {1000, 0} {1000, 0} {1000, 0} {1000, 0} {1000, 0}
PID104 {1000, 0} {1000, 0} {1000, 0} {1000, 0} {1000, 0}
PID110 {1000, 0} {1000, 0} {1000, 0} {1000, 0} {1000, 0}
PID113 {1000, 0} {1000, 0} {1000, 0} {1000, 0} {1000, 0}

mix run --no-halt -e System4.main 5 10000000 9000 100
#PID<0.104.0> {30250, 3463} {30250, 7890} {30250, 3023} {30250, 12167} {30250, 3706}
#PID<0.113.0> {43215, 5243} {43215, 10517} {43215, 5734} {43215, 14616} {43215, 7104}
#PID<0.110.0> {31142, 3561} {31142, 8120} {31142, 3212} {31142, 12265} {31142, 3983}
#PID<0.107.0> {39729, 4663} {39729, 9770} {39729, 4954} {39729, 13909} {39729, 6432}
#PID<0.116.0> {34735, 4201} {34735, 8861} {34735, 3898} {34735, 12584} {34735, 5190}
```

Docker

```
MAIN=System4.main_net PEERS=5 MAX_BROADCASTS=1000 TIMEOUT=3000
LPL_RELIABILITY=50 docker-compose -p da347 up
Attaching to container2, container1, container5, container3, container4, container0
PID110 {1000, 519} {1000, 492} {1000, 516} {1000, 508} {1000, 501}
PID110 {1000, 490} {1000, 497} {1000, 513} {1000, 495} {1000, 505}
PID110 {1000, 497} {1000, 489} {1000, 486} {1000, 501} {1000, 517}
PID110 {1000, 513} {1000, 488} {1000, 505} {1000, 504} {1000, 500}
PID110 {1000, 506} {1000, 503} {1000, 492} {1000, 495} {1000, 496}


MAIN=System4.main_net PEERS=5 MAX_BROADCASTS=10000000
TIMEOUT=9000 LPL_RELIABILITY=0 docker-compose -p da347 up
Attaching to container2, container1, container4, container5, container3, container0
PID166 {26673, 0} {26673, 0} {26673, 0} {26673, 0} {26673, 0}
PID172 {27036, 0} {27036, 0} {27036, 0} {27036, 0} {27036, 0}
```

```
PID182 {26221, 0} {26221, 0} {26221, 0} {26221, 0} {26221, 0}
PID178 {25979, 0} {25979, 0} {25979, 0} {25979, 0} {25979, 0}
PID110 {26519, 0} {26519, 0} {26519, 0} {26519, 0} {26519, 0}

MAIN=System4.main_net PEERS=5 MAX_BROADCASTS=10000000 TIMEOUT=9000
LPL_RELIABILITY=100 docker-compose -p da347 up
Attaching to container1, container2, container5, container4, container3, container0
{25322, 4875} {25322, 5500} {25322, 4849} {25322, 5366} {25322, 4731}
{25209, 4855} {25209, 5484} {25209, 4818} {25209, 5343} {25209, 4708}
{25111, 4835} {25111, 5467} {25111, 4798} {25111, 5325} {25111, 4685}
{25934, 4990} {25934, 5623} {25934, 4976} {25934, 5481} {25934, 4863}
{25081, 4833} {25081, 5464} {25081, 4790} {25081, 5316} {25081, 4677}
```

Chapter 5

System 5

For system 5 we have to set a way to kill a (or some) process at a particular time. The way we do this is we initalize a map in system5.ex where we just set mappings of faulty processes and when they are going to die, For example

```
# Process 3 is going to die in 5 milliseconds
going_to_die = %{3 => 5}
```

Then we pass to every peer when they are going to die (:infinity is an option of course - which means it will never crash) and then we spawn a separate process in peer that just waits for that time and then kills all the components of the peer using Process.exit(pid, :kill) We thought this way reduces friction and binding between the different components of the system.

```
mix run --no-halt -e System5.main 5 1000 3000 50
PID117 {1000, 499} {1000, 462} {1000, 24} {1000, 497} {1000, 516}
PID113 {1000, 496} {1000, 501} {1000, 37} {1000, 511} {1000, 500}
PID125 {1000, 504} {1000, 501} {1000, 32} {1000, 485} {1000, 500}
PID129 {1000, 496} {1000, 477} {1000, 32} {1000, 511} {1000, 514}

mix run --no-halt -e System5.main 5 1000 3000 100
#PID<0.104.0> {1000, 1000} {1000, 1000} {1000, 24} {1000, 1000} {1000, 1000}
#PID<0.116.0> {1000, 1000} {1000, 1000} {1000, 23} {1000, 1000} {1000, 1000}
#PID<0.108.0> {1000, 1000} {1000, 1000} {1000, 24} {1000, 1000} {1000, 1000}
#PID<0.120.0> {1000, 1000} {1000, 1000} {1000, 23} {1000, 1000} {1000, 1000}
```

Chapter 6

System 6

In system 6 we added an eager reliable broadcast mechanism between app and Beb.

EXTENSION: we also added the choice for a lazy reliable broadcast. One can test this by setting lazy = true in system5.ex