Developing an IoT System

ELEC50009 Information Processing

I. Mohamed, T. Moores, A. Rehman, H. Solomon

Table of Contents

Aim

The aim of this project was to recreate the popular arcade game Pong through which users could connect to the game via their DE10 FPGA boards. The data from the Accelerometer SPI interface of each FPGA would be processed locally and used to move the player's paddle through communication with the cloud-based Game server, which would handle updates to the game state.

Design

Functional & Non-Functional Requirements

For the system to perform as stated in the in line with system requirement, the following functional and non-functional requirements need to be met:

- Functional
    - Establish connection between nodes (user FPGAs) and the cloud based server.
    - Have a method of sourcing data to alter players' paddle position via the Accelerometer SPI Interface.
    - Authentically recreate the game 'Pong'.
    - Have means for at least 2 players to join the gain simultaneously.
- Non-functional
    - Have a pleasant user interface.
    - Display the score of the game via the 7-segment display
    - Have minimal Latency between when the user moves their FPGA board and the paddle moves on screen.
    - Have a low-maintenance and stable server application.

System Design & Architecture

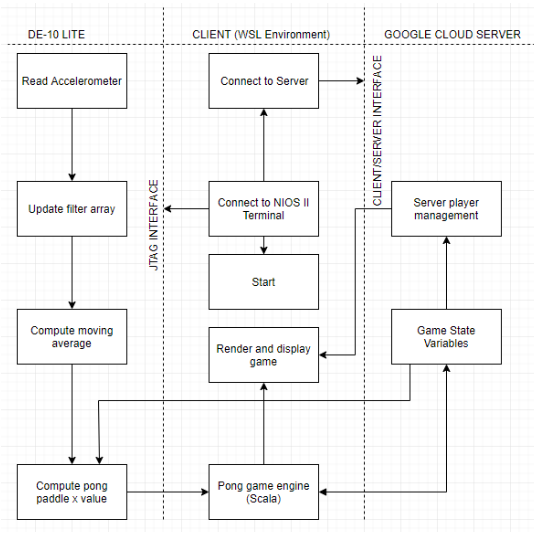


Figure 1. System Architecture

Server

Erlang, a highly specialised programming language designed for building and monitoring multi-nodal networks and distributed applications, was chosen to implement the server to take full advantage of the Open Telecom Platform, which offers a stable connection between nodes of distributed systems with powerful message passing and monitoring features built in[1], as well as the crash tolerance and supervision system which would allow our application to automatically come back online following a crash. Whilst writing the game itself in a native language such as Rust or C++ and calling it from Erlang using ‘Native Implemented Functions’ would have provided a slight performance gain[2], these performance gains did not offset the risk of crashing the entire system such that it had to be manually restarted in the event of the game crashing.

As the server was set up as an OTP node, the clients would also need to be set up as OTP nodes to make full use of the benefits of the platform. While Erlang could have also been used to implement the client, the IO modules are not well suited for a console based game and the immutability of objects means making minor changes to the buffer would have used a larger amount of computing power than was necessary. As the Java Virtual Machine’s console handling is better suited to our needs while also being an OS and system architecture independent platform such that we could run the client on both Windows and Linux, the JVM implementation of Scala was chosen to gain access to its powerful “Process” library to control nios2-terminal and its pattern matching abilities to more easily process the messages received from the server.

FPGA

The C code would need to check the accelerometer value, process it, then send the resulting value to the client using the JTAG connection. It would also have to check for input from the client to update the speed of the paddle to match that of the ball, as well as updating the score on the built in 7-segment displays.
The position of the paddle would be calculated using the angle of the accelerometer to calculate the rate of change of the paddle: a greater angle would result in faster movement up or down. These readings would then be filtered and smoothed to give the impression of real movement of a paddle. The filter was a size 128 equally weighted moving average filter with weights $W_n = \frac{1}{128} \forall\, n \in [0, 127]$. After further review it was clear that this could be heavily simplified, by keeping a running average where the last value in the array is subtracted and the new value is added to an accumulator, resulting in only 2 additions per calculation rather than 128.

Implementation

Approach

The following approach was taken to implement the design:

1. Create a Google Cloud Platform server which could handle connections between clients.
    - Test this by having a simple UI-less program which could be interacted with by multiple users
2. Create a UI on the client program which would aesthetically display the data received from the server
    - Test this by capturing key-presses with an AWT Frame & sending this to the server as the paddle movements
3. Create the desired filter to smooth the movement of the paddle in C.
4. Alter the client to accept data from the FPGA rather than the keyboard
5. Test the final design

User Interface & Cloud Server Implementation

User Interface

As stated in the design brief, all IoT systems via the FPGA will have to receive and process accelerometer data, hence to sufficiently test and optimise the game interface whilst this code was being developed, we created a temporary method to obtain input data in the form of an AWT frame with a key monitor. Pressing the up key would move the paddle up by a given amount & the down key would move the paddle down by the same amount. After the new position of the paddle was calculated, a message was sent to the client application relaying the change in position.

On receiving an output from the FPGA, the client would send a message to the server in the form of an Erlang tuple containing the node name of the client, which allowed the client to be uniquely identified, and the data received from the FPGA. On receiving a new game state from the server, it would update the buffer before clearing the console and printing it to the screen. This gave the illusion of the contents of the console changing. The other messages received from the server were communicated straight to the board once they had been translated into the appropriate format: a control code followed by the score or new speed of the ball.

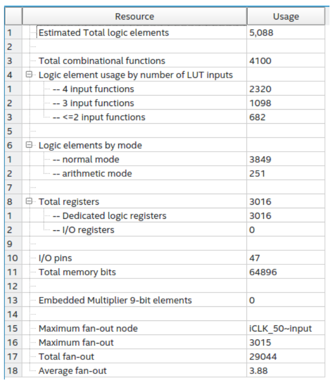
Server

The game's server is comprised of a standard Erlang application[3]. The supervision tree comprises of 2 modules: 'pong game' and 'pong server'.

- 'Pong server' deals with network communication with players as well as controlling the tick rate of the game, which implemented with a timer which is created on load. It receives messages from the timer and clients and acts accordingly depending in the form of the message; each message will be processed by performing an asynchronous cast to the game or, in the case of a tick from the timer, with a synchronous call to the game to get the game state which is then sent to the player, followed by a cast to update the game state. To reduce the potential for malicious attacks on the server, the origin of a tick is confirmed to be the timer as a unique reference is sent along with the tick message, and the node name of each player is known only to that player and the server.
- 'Pong game' is implemented with the 'Generic Server' behaviour, keeps track and updates track of the game state. It updates the game state by moving the ball and players then checking for goals and bounces following asynchronous casts from the Server module and sends the game state to the server following synchronous calls. When it updates the game state, it applies safeguards intended improve playability, such as not allowing the user to move out of the game buffer. The Game module also contains functions to rotate the game state such that each player is sent the game state as though they are on the left. This makes it easier to implement the client and board systems as they can always assume that any local systems only modify the state of the player on the left wall.

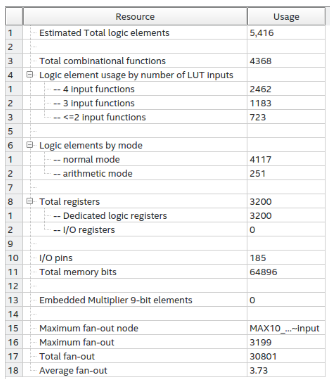
Creating the filter for the accelerometer data

As programming in C was deemed to be more convenient than highly modifying the Quartus project each time the FPGA code was modified, the design of the system in Quartus was kept relatively simple, the only notable exception when compared to the system from Lab 3 being that off chip memory was allocated to allow for a larger C program to be stored and run on the board. This is reflected in the resource usage, which was slightly decreased from that in lab 4.
As the FPGA needed to attempt to get input from the client without knowing whether any was ready, the `O_NONBLOCK` flag was necessary when opening the read/write buffer to prevent `getc` from blocking when no data was available.



	Resource	Usage
1	Estimated Total logic elements	5,088
2		
3	Total combinational functions	4100
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	2320
2	-- 3 input functions	1098
3	-- <=2 input functions	682
5		
6	Logic elements by mode	
1	-- normal mode	3849
2	-- arithmetic mode	251
7		
8	Total registers	3016
1	-- Dedicated logic registers	3016
2	-- I/O registers	0
9		
10	I/O pins	47
11	Total memory bits	64896
12		
13	Embedded Multiplier 9-bit elements	0
14		
15	Maximum fan-out node	iCLK_50~input
16	Maximum fan-out	3015
17	Total fan-out	29044
18	Average fan-out	3.88

Figure 2. Lab 4 Resource Usage



	Resource	Usage
1	Estimated Total logic elements	5,416
2		
3	Total combinational functions	4368
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	2462
2	-- 3 input functions	1183
3	-- <=2 input functions	723
5		
6	Logic elements by mode	
1	-- normal mode	4117
2	-- arithmetic mode	251
7		
8	Total registers	3200
1	-- Dedicated logic registers	3200
2	-- I/O registers	0
9		
10	I/O pins	185
11	Total memory bits	64896
12		
13	Embedded Multiplier 9-bit elements	0
14		
15	Maximum fan-out node	MAX10_...~input
16	Maximum fan-out	3199
17	Total fan-out	30801
18	Average fan-out	3.73

Figure 3. System Resource Usage

Altering the Server/ UI to accommodate input data from the FPGA

The outputs to the server from the AWT and the JTAG Interface were identical, so changing the values was relatively easy. A `nios2-terminal` process was created and IO captured; the newlines were removed from the output & the resulting data was then formatted & sent to the server.

Testing

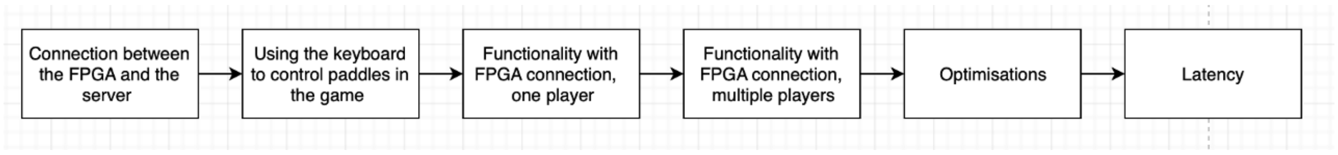
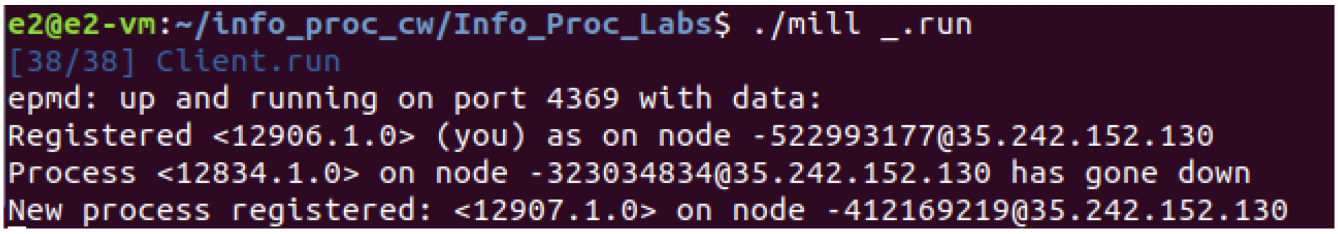

Connection between the FPGA and the server
Using the keyboard to control paddles in the game
Functionality with FPGA connection, one player
Functionality with FPGA connection, multiple players
Optimisations
Latency

Figure 4. Testing workflow

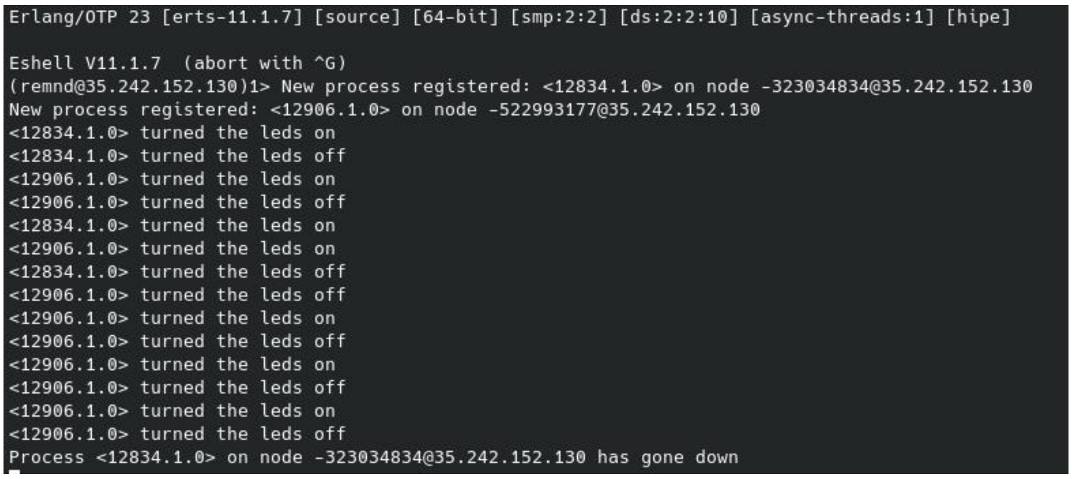
Establishing a connection between the FPGA and the server

A script was written to connect each board to the server, as well as code for the FPGA to send a message when the push button was pressed and released. This allowed for the testing of communication between boards via the simple task of switching the LEDs of all boards on when a button was pressed on any board, then off when that button was released.



```
e2@e2-vm:~/info_proc_cw/Info_Proc_Labs$ ./mill _.run
[38/38] Client.run
epmd: up and running on port 4369 with data:
Registered <12906.1.0> (you) as on node -522993177@35.242.152.130
Process <12834.1.0> on node -323034834@35.242.152.130 has gone down
New process registered: <12907.1.0> on node -412169219@35.242.152.130
```

Figure 5. Local output when testing communcation between boards



```
Erlang/OTP 23 [erts-11.1.7] [source] [64-bit] [smp:2:2] [ds:2:2:10] [async-threads:1] [hipe]

Eshell V11.1.7  (abort with ^G)
(remnd@35.242.152.130)1> New process registered: <12834.1.0> on node -323034834@35.242.152.130
New process registered: <12906.1.0> on node -522993177@35.242.152.130
<12834.1.0> turned the leds on
<12834.1.0> turned the leds off
<12906.1.0> turned the leds on
<12906.1.0> turned the leds off
<12834.1.0> turned the leds on
<12906.1.0> turned the leds on
<12834.1.0> turned the leds off
<12906.1.0> turned the leds off
<12906.1.0> turned the leds on
<12906.1.0> turned the leds off
<12906.1.0> turned the leds on
<12906.1.0> turned the leds off
<12906.1.0> turned the leds on
<12906.1.0> turned the leds off
Process <12834.1.0> on node -323034834@35.242.152.130 has gone down
```

Figure 6. Server output when testing communcation between boards

Using the keyboard to control paddles in the game

As the game runs remotely by design and requires inputs of paddle positions to play, keyboard inputs were used in testing as this allowed for a simple way to move the paddles by a known amount before sending them to the server. This meant that on the client side, key presses were captured and translated into changes to a “position” variable. Testing with one player, some initial bugs were caught, for example the ball going off the screen would sometimes cause an "list index out of bounds" error. Testing with multiple players then allowed the rotation of the board and the rendering of other players to be tested, with the rotation tweaked as necessary.

Testing the FPGA input data

Testing of the FPGA's output data initially comprised of manually observing the data in nios2-terminal. This allowed us to ensure that the values changed as intended as the ball moved before connecting the board to the server. After this was confirmed, nios2-terminal was run from the client and the data set on to the server. A minor fix was required on windows at this stage, as newlines appeared as [10,13] rather than [10], which made the paddle jump to position 13 every other tick.

Performance Metrics

As both the FPGAs and the server sent data at a constant rate, it was unnecessary to ascertain the exact data processing time, so long as it was ensured that the data was received at the predefined rate. Testing to ensure that this rate was as expected involved adding a variable to the client code which measured the time between the receipt of each message. For the server this was 25ms, exactly as expected, however the board appeared to run much slower than expected; ticking once every 100ms, however as this was still a consistent rate and increasing the rate decreased the stability of the system; this value was kept at 100ms.

This meant our main quantitative metric was the latency between sending a message and getting a receipt from the server or the board. To test this for the server, we tested the time taken to connect to the server, which came to 120-150ms on average, and the time taken to register as a player on the server and receive confirmation, which took around 20-30ms on average, which were both very acceptable figures. For the FPGA, the round-trip time came to consistently 100ms. As this was the same as the slower-than-expected rate at which messages were received from the board, this is likely a lower limit of the transmission rate of the board and as such the latency of the messages arriving at and being received from the board could not accurately be measured as one or both of these journeys had a disproportionate delay.

Optimisations

After assessing the functionality, the next stage in the testing process was to optimise user experience. The initial ball speed was set at 2 bits per 25ms (It would move 2 bits diagonally in the 256 x 256 display buffer every server tick). Upon running the game, it was found this was to fast for the game to be playable. After further testing, this value was tuned to be 2 bits per server tick. The user paddles on screen was initially set to be 10 bits long, yet after several iterations of running the game this value was again tuned to be 25 bits. The skew of the ball as it bounced off a paddle was also determined by trial and error until an acceptable value was found.

Conclusion

In conclusion, the final implementation of this project resulted in an IoT system that fulfilled all coursework requirements alongside the functional & non-functional requirements needed to recreate the Pong arcade game authentically. The users paddle movement is determined through data from their DE10 FPGA board, which was smoothed and filtered using a moving average calculation. This information would then be communicated from each node to the cloud-based server which would update the player's position and send the user a game state to display the on the console-based user interface each tick. Possible extensions of this design could improve be to improve filtering, perhaps by implementing an anomaly detection function to prevent unexpected jumps in data causing unexpected movement of the paddle, as currently these have the potential to affect the position if they are sufficiently large. The user interface could be improved through making it more friendly towards 3+ player games, perhaps by changing the shape of the user interface to an N sided polygon, where N is the number of players connected. This would improve the game by eliminating unfair advantages from the wall a player is allocage, eg. in games with 3 players on a 4 sided board where 1 player is opposite a wall while the other 2 are facing each other, wherein it is harder for the third player to score goals than the players who are opposite each other.

Appendix

When the script is run in the users' terminals, a connection between that node (FPGA) and the server is established and the game appears in the terminal taking up the whole window. The buffer size scales automatically to the size of the users' terminals, allowing the user to play the game at any resolution. The walls and paddles are represented by hashes and the ball as an 'O'.

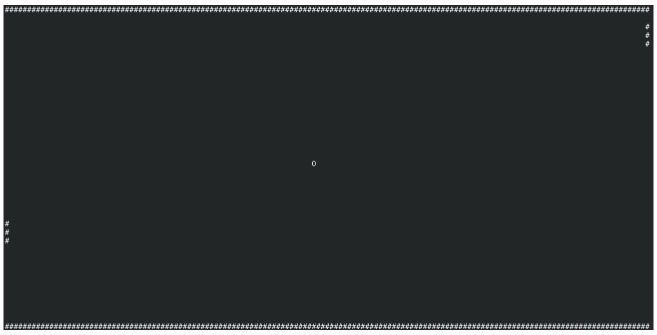


Figure 7. Playing in a small terminal

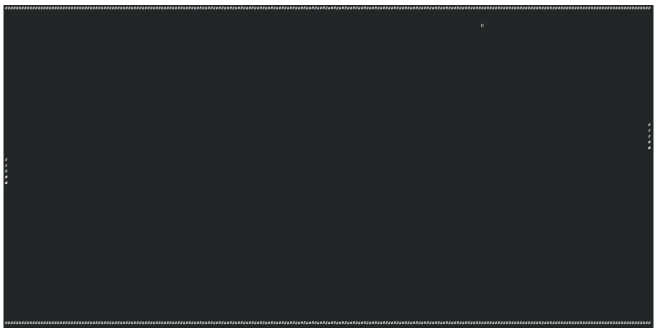
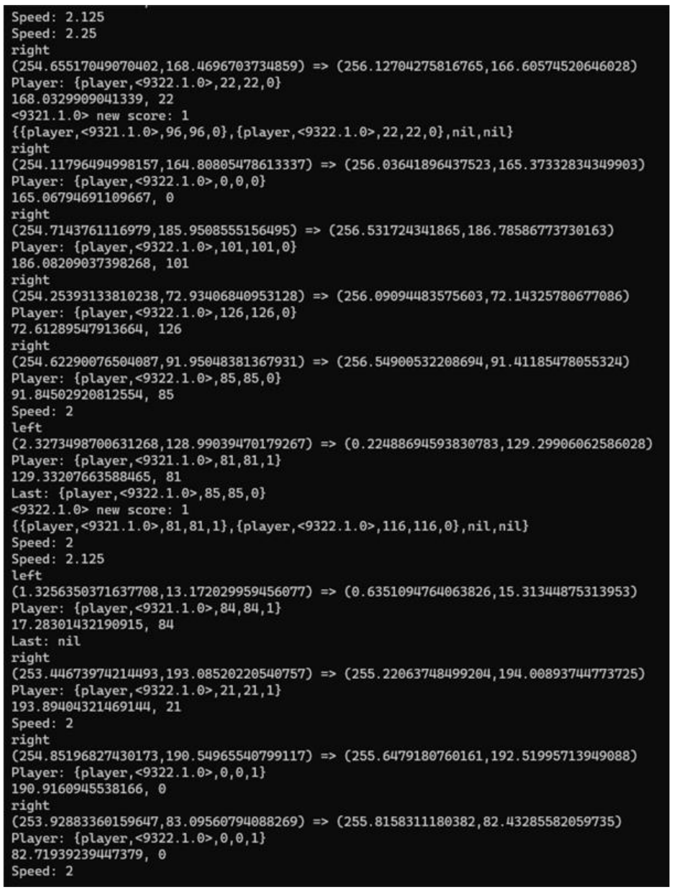


Figure 8. Playing in a large terminal

The server then begins to manage game play by tracking the coordinates and the speed of the ball, as well as the player who last hit the ball. If the ball crosses the goal and a player is the last to hit it, as opposed to the wall (nil in figure), then that players' score is updated. The ball starts off with a speed of 2 and increments by 0.125 every time it hits a wall or paddle, and resets when the ball is missed.



```
Speed: 2.125
Speed: 2.25
right
(254.65517049070402,168.4696703734859) => (256.12704275816765,166.60574520646028)
Player: {player,<9322.1.0>,22,22,0}
168.0329909041339, 22
<9321.1.0> new score: 1
{{player,<9321.1.0>,96,96,0},{player,<9322.1.0>,22,22,0},nil,nil}
right
(254.11796494998157,164.80805478613337) => (256.03641896437523,165.37332834349903)
Player: {player,<9322.1.0>,0,0,0}
165.06794691109667, 0
right
(254.7143761116979,185.9508555156495) => (256.531724341865,186.78586773730163)
Player: {player,<9322.1.0>,101,101,0}
186.08209037398268, 101
right
(254.25393133810238,72.93406840953128) => (256.09094483575603,72.14325780677086)
Player: {player,<9322.1.0>,126,126,0}
72.61289547913664, 126
right
(254.62290076504087,91.95048381367931) => (256.54900532208694,91.41185478055324)
Player: {player,<9322.1.0>,85,85,0}
91.84502920812554, 85
Speed: 2
left
(2.3273498700631268,128.99039470179267) => (0.22488694593830783,129.29906062586028)
Player: {player,<9321.1.0>,81,81,1}
129.33207663588465, 81
Last: {player,<9322.1.0>,85,85,0}
<9322.1.0> new score: 1
{{player,<9321.1.0>,81,81,1},{player,<9322.1.0>,116,116,0},nil,nil}
Speed: 2
Speed: 2.125
left
(1.3256350371637708,13.172029959456077) => (0.6351094764063826,15.31344875313953)
Player: {player,<9321.1.0>,84,84,1}
17.28301432190915, 84
Last: nil
right
(253.44673974214493,193.08520220540757) => (255.22063748499204,194.00893744773725)
Player: {player,<9322.1.0>,21,21,1}
193.89404321469144, 21
Speed: 2
right
(254.85196827430173,190.54965540799117) => (255.6479180760161,192.51995713949088)
Player: {player,<9322.1.0>,0,0,1}
190.9160945538166, 0
right
(253.92883360159647,83.09560794088269) => (255.8158311180382,82.43285582059735)
Player: {player,<9322.1.0>,0,0,1}
82.71939239447379, 0
Speed: 2
```

Figure 9. The server logs recording when the ball contacts a paddle, goal or wall

References

[1] Fred Hebert, "The Hitchhiker’s guide to concurrency", in *Learn You Some Erlang For Great Good!* [Online]. Available at: https://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency#concepts-of-concurrency. [Accessed 29 March 2021].

[2] Ericsson AB, Stockholm, Sweden. "erl_nif" in *Erlang Run-Time System Application (ERTS): Reference Manual, Version 11.2*. [Online]. Available at: https://erlang.org/doc/man/erl_nif.html. [Accessed 29 March 2021].

[3] Ericsson AB, Stockholm, Sweden. "Erlang — Applications" in *OTP Design Principles: User’s Guide, Version 11.2*. [Online]. Available at: https://erlang.org/doc/design_principles/applications.html. [Accessed 29 March 2021].