The Actor Model, Part Three

CSCI 5828: Foundations of Software Engineering
Lecture 15 — 10/11/2016
Goals

• Cover another example of using processes in Elixir
  • Taken from the following book
    • The Little Elixir & OTP Guidebook by Benjamin Tan Wei Hao
    • Published by Manning (last week!)
• Introduce the ability to run processes on multiple nodes
Retrieving Temperature Information

• Present one more basic example of Elixir processes
  • This program can be used to look up the temperatures of an array of cities
• The code is deployed in a mix project
  • I can't distribute this code but it can be downloaded from Manning
  • Let's review the code now.
• To run this project, we invoke the interpreter with the following command
  • `iex -S mix`
• and then enter the following at the prompt
  • `cities = ["Singapore", "Monaco", "Vatican City", "Hong Kong", "Macau"]`
  • `Metex.temperatures_of(cities)`
Discussion

• Once again, the solution is strikingly straightforward
  
  • A worker takes care of making a web service call to retrieve the information for a single city
  
  • A coordinator takes care of waiting for all the responses
  
  • A client function takes care of creating the workers and telling the coordinator how many responses to accept
    
    • In this program, the coordinator prints out the results when all of them have been received
  
  • The solution developed by an entirely different author is still very much in line with the design we saw last week with the Fibonacci calculator
Nodes and Distribution

- The Erlang virtual machine is used to execute Elixir programs
  - In an analogous way that Clojure programs compile down to Java bytecodes and are executed by the Java Virtual Machine
- One cool feature of Erlang virtual machines is that they have the capability to act as nodes that can form clusters
  - Elixir actors running on one node can easily route messages to actors running on other (possibly) distributed nodes
- To set this up in Elixir, you can launch iex and give it a node name
  - For security reasons, you also give it a “cookie”; only nodes with the same “cookie” can talk to one another
    - `iex --sname node_one --cookie jiriki ← can be any string`
Connecting Nodes

• Once you have launched a node, you need to tell it about the other nodes
  • `iex --sname node1 --cookie jiriki`
  • `iex --sname node2 --cookie jiriki`

• Checking status
  • `node1> Node.self => :"node1@<domain_name>"`
  • `node2> Node.self => :"node2@<domain_name>"`

• Connecting
  • `node1> Node.connect(:"node2@<domain_name>")` => true

• Both nodes are now connected to each other
  • `node1> Node.list => [:"node2@<domain_name>"]
  • `node2> Node.list => [:"node1@<domain_name>"`
Sending Code Between Nodes

• Let’s define a function
  • `node1> whoami = fn -> IO.inspect(Node.self) end`

• And send it to another node to be executed
  • `node1> Node.spawn(:"node2@<domain_name>", whoami)`
    • `node1 REPL prints: node2@<domain_name>`

• Pause to think about what we just did and how easy it was
  • We just
    • defined a function
    • sent it over to another machine as data
    • that machine converted the data back to a function
    • executed it
    • sent back the result
    • and our original machine then displayed the result
Ticker: Client-Server Example (I)

• Our book now delves into a simple client-server example

• The server is a program that generates notifications every two seconds
  • It provides a method that allows clients to subscribe to its events

• The client is a simple program that registers with the server and prints out a message for each event
  • Let's review the code now

• Note: when a client sends its pid to a different machine, it automatically gets translated into a pid that refers back to it on the other machine
  • no need for your code to worry about details like that! :-)

© Kenneth M. Anderson, 2016
Ticker: Client-Server Example (II)

• To run the example
  • launch two iex servers named node1 and node2 using the same cookie
  • connect the nodes together
  • compile ticker.ex in each of them
  • In node1, start up the server and client
    • Ticker.start
    • Client.start
  • In node2, start up the client
    • Client.start
  • Watch the messages fly across the screen! :-)

© Kenneth M. Anderson, 2016
Input/Output and Nodes (I)

• In the Erlang VM, input and output are handled by I/O servers that are implemented as processes

• As with all processes, they have an associated pid
  
  • we can communicate with these servers via this pid
  
  • not exactly

  • we pass the pid to a function called map_dev
  
  • that function returns a device that is then used to perform I/O

• You can get the default device (i.e. standard output) of an Erlang VM by calling the function :erlang.group_leader()
Input/Output and Nodes (II)

• With this as background, we now have what we need to pass character data from one VM to another
  • Or to write output from one node to a file (i.e. a "device") on another node

• Watch
  • start node1 and node2; connect them
  • Now, associate standard out of node2 with a global name
    • :global.register_name(:two, :erlang.group_leader)
  • Retrieve that name using the "whereis" function on node1
    • two = :global.whereis_name :two
  • Send data from node1 to standard out on node2
    • IO.puts(two, "Hello, "); IO.puts(two, "World!")
Wrapping Up

• We saw another basic example of processes in action
  • this time retrieving temperature information from a web service
  • each web service call can take a different amount of time to process
    • Elixir processes make it easy to deal with that uncertainty
    • We simply tell our coordinator how many responses to expect and then wait for them to arrive
• We then took a look at the material from our textbook on distributing processes across nodes
  • we ran the nodes on the same machine but the examples would have worked just the same if the nodes ran on different machines
    • what's remarkable is how easy it was to create a distributed program using this paradigm