NOTE: Hand-in instructions are at the end of this document. It is very important you follow these instructions. Failure to do so might result in receiving 0 on this lab.

DUE: Februay 14, 2025 9:00pm ET

Dependencies and Installation

We begin by talking about dependencies required to do the lab, and setting your computer up for this.

Git

We use Github and Git for this class. Even bootstrapping for this class requires Git. While we assume you have sufficient exposure to Git, the following links might come in handy:

- Eddie Kohler's guide to Git Github Guides: Hello World
- Github Guides: Git Handbook

Installing Elixir

We will be using Elixir in this lab. You can set up Elixir on your computer if you desire. To do so please follow the instructions on the Elixir website. If you go this route, you are responsible for figuring out how installation works. Please make sure you have `Erlang/OTP 23` and Elixir version `1.10.4` installed. In particular, this means that when installing on Debian or Ubuntu you should use the Erlang solutions repository as suggested on the Elixir webpage. (If that last sentence makes no sense to you, and you use neither Ubuntu nor Debian nor a derivative, then don't worry about it. Computers can be needlessly complicated at times, and this bit is not important for anything we will talk about in class.)

On Editors

We expect you already have a favorite editor, and recommend you use that. Elixir has plugins for almost every editor, and you can find them using Google. I (Panda) use a combination of Vim and Emacs configured to act like vim. I am not sure why I made these choices, and feel people should make their own choices on this matter.

However, if you are looking for a recommendation, I found that VSCode with the vscode-elixir plugin seems to work well, and given the popularity of VSCode that is what I would recommend for simplicity.

You do not need to install VSCode within the VM, instead you can install it locally and then follow the steps in this blog post to connect VSCode to the VM. Remember that you need to explicitly enable vscode-elixir after you connect to the VM using the SSH plugin.

Getting started with Elixir

Elixir is a **functional**, **actor based** language that runs on the Erlang virtual machine (BEAM). We chose it for this class since the programming model closely resembles the process based description and model used in much of the readings in the class. That said, we expect that it will take a bit of effort to get used to this language, and the first lab is designed to help with this.

To start with, you should work through the first 9 sections of the Elixir Getting Started guide, that is from Introduction – Recursion. You don't need to go through any of the rest. You should not using any of the OTP for this class. What follows assumes you have already read through the first 9 sections of the getting started guide.

Once you are done, you are ready to work on the lab.

Getting the Lab

We are using Github classroom to distribute labs throughout this semester. To create a repository for Lab 1 go to the URL https://classroom.github.com/a/8fd82hTz after logging into Github. This will present a button you can use to accept the assignment, which in turn will create a repository for you under the `nyu-distributed-systems`.

For example, my Github username is `apanda`, and going to that website allowed me to create a repository at `https://github.com/nyu-distributed-systems/sp25-lab1-apanda`. Your repository is private by default, and is only accessible to you, and the teaching staff. Please do not attempt to change this.

The Lab Itself

The project description that follows, while correct, is better presented in the Markdown README included with the template code. We suggest switching to that version now.

The links in this instruction are to pages rather than within pages. This is due to some issues with URL encoding in the Markdown tool (Pandoc) used when generating this document. The README file in the Lab template is both easier to read and more authoritative. We recommend using that file rather then the PDF. However at present both have identical content.

- You should work through the first 9 sections of the Elixir Getting Started guide, that is from Introduction – Recursion. You don't need to go through any of the rest. You should not using any of the OTP for this class. What follows assumes you have already read through the first 9 sections of the getting started guide.
- Create the repository using Github classroom and clone it either on your computer or the VM you set up. You need to clone it wherever you installed Elixir.
- You need to first install all the project dependencies. To do so run the following command from the repository root:

```
> mix deps.get
> mix
```

This process might take a bit of time (several seconds) as the code gets built. You should also see several warnings: as you work on the project these warnings will disappear.

- Test to make sure things look correct. Again from the project root run

```
> mix test
```

You should get output of the following form:

“‘

==> emulation .....

Finished in 0.2 seconds 5 tests, 0 failures

Randomized with seed 567507 ==> lab1

LossfreeCounterTests

- test check that repeated gets work work (0.00ms)

  1) test check that repeated gets work work (LossfreeCounterTests) apps/lab1/test/lossfree_counter_test.exs:38
     (EXIT from #PID<0.325.0>) an exception was raised: (RuntimeError) Not implemented (lab1 0.1.0) lib/intro_lab.ex:63: IntroLab.lossfree_counter/1

- test Check that decrements work (0.00ms)

  2) test Check that decrements work (LossfreeCounterTests) apps/lab1/test/lossfree_counter_test.exs:70
     (EXIT from #PID<0.336.0>) an exception was raised: (RuntimeError) Not implemented (lab1 0.1.0) lib/intro_lab.ex:67: IntroLab.lossfree_counter/1

- test check that increments work (0.00ms) ...

  “‘

  Again, you will fix these tests as you work through the lab.

- You are now ready to work on the lab. All of the code you need to edit is in `apps/lab1/lib/intro_lab.ex`. Open this in your favorite editor.

Looking through the code

- When you open `apps/lab1/lib/intro_lab.ex` the first line defines a new module called `IntroLab`. In Elixir (and Erlang) modules are a unit of organizing code and can consist of functions, macros and other things.
- The next line (`import Emulation, only: [...]`) loads the Emulation layer used by this class. All of the labs will include such a line, we overwrite the default Elixir implementation of `spawn`/2 and `send`/2 to inject failures and delays.

> An aside on how functions in Elixir are specified: `spawn/2` means that we are talking about a function named `spawn` that accepts 2 arguments. This might of course be distinct from `spawn/3` which is a function of the same name which accepts 3 arguments.

- The next line `import Kernel, except...` is the next part of emulation setup, it makes sure that we do not import any of the `spawn` functions or the `send` function defined by Elixir itself.

> All of the Labs in this class will include the previous two lines. This is what allows us to emulate an asynchronous network on a single host.

- You can now skip ahead to the line with `@moduledoc`. `@moduledoc` is provided a string that documents the module you are looking at. You should read through the moduledoc for any template code we provide since they include instructions on how to work through the lab.

Your First Distributed System: A Counter

Learning Objectives

The main things you need to learn from this part of the lab is

- Recursion and how state works in Elixir.
- Running and testing code.
- Build your first distributed system.

Desired Semantics

You are going to construct an integer counter process which works as follows:

- When the process starts it initialize an integer referred to as the counter from here on with the value 0.
- When the process receives a message `:increment` it increases the counter value by 1.
- When the process receives a message `:decrement` it decreases the counter by 1.
- When the process receives a message `:get` it sends the sender a response with the current value of the counter.

Code Walk Through

In Elixir it is common to use atoms as a way to decide what a message should do. Most Elixir code explicitly writes out the atoms, and constants are rare. However. pedagogically (and for testing) assigning simple constants to atoms makes things a bit easier. In our code we will often abuse Elixir's macro mechanisms to construct such constants. This section includes three:

- `@inc` which maps to `:increment`

- `@dec` which maps to `:decrement`
- `@get` which maps to `:get`

The logic executed by the counter process (we will return to the question of how this process is created shortly) is specified by the `lossfree_counter/0` function. Let us look at a few things above this function:

- First note that this function is exported, i.e., it can be called from outside the module. This is because we use `def lossfree_counter` to define this function. See Modules and function in the Elixir tutorial if this seems strange. You should neither remove nor change the function signature for any **exported** functions in this class. Such functions are considered a part of the interface exposed and should not be changed.
- Since this is a public (exported) function we can provide documentation about it. This is done using the `@doc` string above. You **should** write such documentation for your own code too, recording anything you would want your future self to remember if you need to change the code in the future. If you want to read the documentation in a formatted manner do the following: "'

> ## Go to the apps/lab1 directory
>
> mix docs # This will build your docs
>
> ## Now if you go to build/index.html you will have you

```
# code laid out in pretty HTML.
```

```

* We also specify the arguments and return value of the function using the
[`@spec`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L43) line. Elixir does not enforce the `@spec` line but there is
tooling which can use this for analysis. It is also used by the documentation
tool to determine what your function accepts, etc. In this case the `@spec`
line says that the function takes no arguments, and never returns (`no_return()`).
See the [documentation](https://hexdocs.pm/elixir/typespecs.html) for other
specs.
* Observe that all the function does is call [`lossfree_counter/1`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L45) with 0
```

```
as an argument. This is a very common pattern in Elixir and other functional
languages, where function arguments and recursion is used to store and update
state.

> Statements such as `a = 5` or function arguments such as `x` in `def add(x)`
> do not define "variables" in functional languages. The statement `a = 5` just
> says that the compiler can replace occurrences of `a` in the next statements
> (within the same scope and until another statement such as `a = 2` occurs) with
> `5`. Similarly, within the body of `def add(x)` occurrences of `x` can be
> replaced by the argument. This is close to, but not the same as, constants in
> other languages. This seems to trip people up when starting. The technical
> difference is that as opposed to many languages, things like `a` and `x`
> in functional languages do not refer to locations in memory but instead to
> values. You should be careful with this difference as you work through the
> labs.

We will skip `lossfree_counter/1`, which is the function you need to fill out for
the moment, and return to it in a little. Let us instead look at
[`test_lossfree_counter/0`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L71) which shows a case where we use the counter process.
Let us look more closely at this function:

* First, please do not change any functions named `test_*` in the lab template code.
We provide these functions to show you how things are meant to be used. You are
**encouraged** to write additional functions of this form if you want, just do not
modify the ones we provide.
* The function [starts](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L72) by calling the `init/0` function in the `Emulation` module.
This is the [emulation module](https://cs.nyu.edu/~apanda/classes/fa20/emdocs/Emulation.html)
used by the class. You should read its [documentation](https://cs.nyu.edu/~apanda/classes/fa20/emdocs/Emulation.html)
if you want to learn more about the module.
```

```
* [Next](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L74) the function [spawns](https://cs.nyu.edu/~apanda/classes/fa20/emdocs/Emulation.html)
  a **process** named `:counter`. The `:counter` function executes
  `lossfree_counter/0`, which we talked about above. The syntax `&lossfree_counter/0`
  is used in Elixir to pass the function (rather than its result) as an argument to
  the `spawn` call.
* [Next](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L75) we use [`send/2`](https://cs.nyu.edu/~apanda/classes/fa20/emdocs/Emulation.html)
  to send messages to the `:counter` process started above.
* Once we send the `@get` message, we use [`receive`](https://elixir-lang.org/getting-started/processes.html)
  to receive a response with the current value of the counter, and return true or
  false depending on whether it is 0 or not.

  > *Note 1:* As we will see shortly `receive` works differently depending on whether
  > it is executed within the emulation environment or not. Code executed within the
  > emulation environment must be called a function running in a `spawn`ed process.
  > Anything else is running outside the emulation environment.

  > *Note 2:* As is common in most functional (and even many imperative) languages,
  > Elixir functions return the result of the last statement executed by a function.
  > As we will discuss next, `after` is a special form, and hence the last real
  > statement executed in this case is `v == 0`, which is what is returned by the
  > function.

* Function execution can fail for a variety of reasons in Elixir. The `after`
  statement is a way to ensure that some code is always run, regardless of
  whether the function completes correctly or not. This is similar to the `finally`
  clause in many other languages. In this function we call
  [`Emulation.terminate/0`](https://cs.nyu.edu/~apanda/classes/fa20/emdocs/Emulation.html)
  which clears up emulation state. As we note above statements executed in the
  `after` clause act as if they were run **after** the function was done executing,
   and do not alter the return value.
```

```

Now that we have seen this function, let us try running it to see what
    happens. To do this, go to `apps/lab1` and:
```

> mix run -e ‘IntroLab.test_lossfree_counter()’

```

This should produce output similar to:
```

(EXIT from #PID<0.94.0>) an exception was raised: (RuntimeError) Not implemented (lab1 0.1.0) lib/intro_lab.ex:54: IntroLab.lossfree_counter/1

```

OK, so we need to implement things to fix this.

In the above, `mix run -e` takes an Elixir **expression** and executes
    it in the
current application. For example:
```

> mix run -e ‘IO.puts(1 + 1)’ 2

```

In the snippet above we had `mix run -e` compute and then print (using
[`IO.puts`](https://hexdocs.pm/elixir/IO.html)) the result of adding 1
    and 1.

Elixir also offers an interactive REPL (read-eval-print-loop) that you
can use for running and testing things. To use the REPL run:
```

> iex -S mix Erlang/OTP 23 [erts-11.0.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [hipe] [dtrace]

Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help) iex(1)> IntroLab.test_lossfree_counter() **(EXIT from #PID<0.215.0>) shell process exited with reason: an exception was raised:** (RuntimeError) Not implemented (lab1 0.1.0) lib/intro_lab.ex:54: IntroLab.lossfree_counter/1

Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help) iex(1)> 13:33:55.816 [error] Process #PID<0.223.0> raised an exception ** (RuntimeError) Not implemented (lab1 0.1.0) lib/intro_lab.ex:54: IntroLab.lossfree_counter/1

nil iex(2)>

```

## Implementing the Code

Now we can turn to implementing your code, and completing this task. To
    do so
you will need to add code to [`lossfree_counter/1`](https://github.com/
    nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/
    intro_lab.ex#L49), which (from above) is called
```

```
by `lossfree_counter/0`. In this function observe that:

* There is no `@doc` string, this is because it is a private (`defp`)
  function, and Elixir does not allow `@doc` strings.
* We begin by waiting to `receive` a message using [`receive do`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L50).
* We then determine what type of message we have received.
  > As we noted above, `receive` within an emulated process behaves differently
  > from the outside. Inside a process, all messages received are tuples of the
  > form `{sender, message}`. This is because in Panda's experience, you end up
  > almost always wanting the `sender`, and rather than have each of you struggle
  > through this adding it in the framework was easier.
* For the `@inc` and `@dec` messages we match on `{_, @inc}` and `{_, @dec}`. The
  `_` in this case indicates that the code **does not** use the sender. From the spec
  above, we don't send messages for `@inc` and `@dec`.
* In the code for `@inc` you can see that we [`raise`](https://elixir-lang.org/getting-started/try-catch-and-rescue.html)
  a `Not implemented` error, which is indeed what you ran into above. * You should
  replace this `raise` with your code.
* To see how this might work look at the `@get` code, which does not change the
  counter but sends its current value and [recurses](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L62) so
  the function continues running. In the `@inc` and `@dec` case you should not
  send a message, but should recurse after changing the value appropriately.

### Print debugging

We are just briefly going to consider how you might debug a problem. For example
consider the case where you want to print the value sent by `@get` and its receiver.
To do so you can change the `{sender, @get}` branch to add the following code anywhere
**before** the recursive call to `lossfree_counter/1`:
```

IO.puts("Sending #{inspect(sender)} value #{current}")

```

There are several things to consider in this line:
```

```

* `IO.puts`, as we noted above, prints to standard out.
* Elixir strings support [interpolation](https://hexdocs.pm/elixir/String.html).
  What this means is that anything that appears within `#{}` in a string is treated
  as code which is executed and its return value is inserted into that part of the
  string. For example, `#{current}` above inserts the value of `current` at that
  location in the string.
* When performing string interpolation, Elixir needs to convert values to
  strings. Some things, e.g., process IDs do not come with a way to do this.
  [`inspect/2`](https://hexdocs.pm/elixir/Kernel.html) is a debugging
  method that works around this. When in doubt you should use **inspect**.

You should try using `IO.puts` in your lab, but you should **remove it** before you
hand your code in. In general, you should minimize the amount of debugging output
you send to us.

### Testing this part

We provide you with tests to check the correctness of your labs. You can run
tests for the entire lab by going to `apps/lab1` and running
```

mix test

```

However, this tests all parts of the lab, and it might be hard for you to discern
whether just the counter portion is correct or not. To help with this we have split
the test code so you can test each portion independently. To test just the counter
run:
```

mix test test/lossfree_counter_test.exs

```

At present there are four tests in that file, if you can pass all 4 you should
be in good shape on this part of the lab.

```

```
> About testing: We will provide you some tests for all of the labs.
   However, you
> should not assume that our tests are sufficient. This is for two
   reasons: one,
> we withhold some tests from you so you cannot overfit your solution
   to our tests;
> second, the risks for your implementation (and hence what should be
   tested) are
> something you understand better than we do. As a result you should
   think of our
> tests as something necessary for correctness, but not **sufficient**.
    You can (and
> should) add your own tests. You should edit `test/
   lossfree_counter_test.exs` to see
> how. Please **submit** any additional tests you add.

### Code formatting and Linting

Formatting your code is useful for readability, and might even help you
    identify
bugs quicker. All labs are set up to provide you tools for automatic
   formatting. To
do so go to `apps/lab1` and run
```

> mix format

```

We also support a linter that can help you both find all TODO's and
   find any code
problems. **We strongly recommend using the linter periodically** and
   fixing all
linting bugs before submission. To run the linter go to `apps/lab1` and
    run:
```

> mix credo –strict

```

# Part 2: Build a Distributed Protocol for Reliable Message Delivery

The instructions and description are less detailed for this part. We
   assume
you will use the skills acquired from the previous part to understand
   and work on
this part of the lab. Reading the code in `intro_lab.ex` is of course
   very useful.

## Learning Objectives

The main things you need to learn from this part of the lab is

```

```
* How to deal with message losses.
* How to use timers.
* How drop probability impacts the number of times messages have to be resent.

## Desired Semantics

In this part of the lab you need to build functions for sending
([`reliable_send`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L106))
and receiving ([`reliable_receive`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L120))
messages over an asynchronous network, i.e., one that can randomly delay or drops messages.

In building this you should try to limit the number of messages you send, specifically:

* `reliable_send` should wait between resending messages, in case its previous attempt
  to send a message succeeds. We inject delays into messages, and this can result in
  situations where signals from the receiver can be delayed.
  In the code [`@send_timeout`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L88)
  represents time in milliseconds that we think you should wait between resends.
* `reliable_receive` should signal the sender when a message is received.
* `reliable_send` should **not** resend any messages after the sender receives the signal from
  the receiver.
* `reliable_send` takes a timeout parameter, and it should stop trying to send the message
  once this timeout has expired.
* `reliable_receive` should have the same return as `receive`, i.e., it should return
  a tuple of the form `{sender, message}`.

Additional, `reliable_send` should either return the number of resend attempts it had to make
before being signaled by the receiver, or the atom `:notok` if the send timed out.

> Some of you might be inclined to build in fancy retry logic. It is not necessary
> in this case, and I recommend going with the simplest possible strategy.

## Setting alarms

```

```
The semantics above require receiving timeouts. The emulation
    environment provides a call for
setting timers, [`Emulation.timer/1`](https://cs.nyu.edu/~apanda/
    classes/fa20/emdocs/Emulation.html),
that can be used to set a timer. You can cancel a previously set timer
    using the
[`Emulation.cancel_timer/1`](https://cs.nyu.edu/~apanda/classes/fa20/
    emdocs/Emulation.html)
call. Below we show how you might use both:
```

defp timer_test do t = Emulation.timer(10) receive do :timer -> # Observe no sender here, because the message is from inside. IO.puts("Timer went off") {sender, m} -> IO.puts("Sender #{inspect(sender)} sent message #{inspect(m)} before timer") case cancel(t) do false -> IO.puts("Timer has already gone off, and there is a :timer msg waiting") n -> IO.puts("#{n} ms remains on the timer") end end end

spawn(:timer_proc, timer_test)

```

You need to use this in your implementation.

## Nonce

Observe that the `reliable_send` function accepts a `nonce` as an
    argument. You
might wonder about why?

Observe that a sender might resend a message **after**
the receiver has sent its response. In this case, if the sender calls
`reliable_send` a second time, it needs to be careful in associating
    acknowledgments
with particular messages.

The nonce gives you a way to do this: we guarantee that callers will
    supply
a unique nonce message. You should use the nonce to handle the case
    described above.

## Emulating Asynchronous Networks

The function [`test_reliable_send_and_receive`](https://github.com/nyu-
    distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/
    intro_lab.ex#L152)
(which you can execute by calling `mix run -e 'IntroLab.
    test_reliable_send_and_receive'` sets up an asynchronous
network to test your code. The operative line that does this is:
[`Emulation.append_fuzzers([Fuzzers.drop(0.2), Fuzzers.delay(10.0)])`](
    https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/
    master/apps/lab1/lib/intro_lab.ex#L157).

Fuzzing is a testing technique that we adopt here for your labs, the
    emulation environment is designed to allow different
```

```
types of fuzzing (and a few other features we might use later). In this case we set the environment up so that packets have
a 20% chance of being dropped, and experience a mean delay of 10ms. We use an exponential distribution for delays. This is
a pretty bad network to be operating in.

## Anonymous functions and closures

The `test_reliable_send_and_receive` function also shows an
[example](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L160)
where we use an anonymous function created with `fn`. You are going to repeatedly
use this pattern: `spawn/2` expects a 0-arity function (i.e., one that takes no
arguments). If you want to run a more complex function you need to create one using
`fn`. Anonymous functions in Elixir are closure, specifically this means that in the
`fn` body all names have the bindings they had when the `fn` was created. You should
play around with `fn`'s to understand how they work.

## Measuring the impact of drops on performance

We have also provided a function
[`measure_pings_at_drop_rate/2`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L224) that
can be used to measure the number of retries it takes to get a packet across
as you change the probability of dropping a packet. The function
[`test_measure_pings/0`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L244)
shows you how this can be used.

**REQUIRED WORK:** As a part of your handin, modify `apps/lab1/README.md` to report the
median number of retries (for 100 trials) when the drop probability is:

* 0.001 (i.e., 0.1%)
* 0.005 (i.e., 0.5%)
* 0.01 (i.e, 1%)
* 0.05 (i.e., 5%)
* 0.1 (i.e., 10%)
* 0.2 (i.e., 20%)
* 0.5 (i.e., 50%)

Also report any conclusions you can draw from these observations.

## Testing
```

```

The unit tests for this portion of the lab are contained in `test/reliable_test.exs` and can be run by calling
```

> mix test test/reliable_test.exs

```

# Part 3: Combining the previous two parts to build a Key Value Store

The final portion of this uses the `reliable_send` and `reliable_receive` functions
you developed previously to create a key-value store. This is a hashmap (or a
dictionary) accessible over a network. Key-value stores such as Redis and Cassandra
are widely used in practice, and here you are going to construct a relatively simple
one.

## Learning Goals

* Use the reliable send and receive protocol to build an application.
* Learn how to use maps in Elixir.

## Desired Semantics

Complete the key-value store in
[`reliable_kv_server/2`](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L259)
so that a process spawned with this function:

* Maintains a hashmap in the `state` argument to `reliable_kv_server/2`.
* Uses the `count` argument as a nonce when using `reliable_send`. Remember that in
  order to do so you must increment `count` every time you use `reliable_send`.
* When it receives a `{@set, key, value}` message, it updates the `state` hashmap
  so that `key` is associated with `value`.
* Responds to `{@get, key}` messages by sending the sender a tuple of the form
  `{key, value}` where `value` is the current value associated with key `key`. If
   no such value exists, return `{key, nil}`.

You should reuse `reliable_send` and `reliable_receive` when working on this part.

## Maps in Elixir

```

```
Completing this part of the project requires that you use Elixir's
[Map](https://hexdocs.pm/elixir/Map.html). We are going to end up using this
in future projects, so it is good to gain some familiarity.

The stencil code already constructs new Map for you in
[reliable_kv_server/0](https://github.com/nyu-distributed-systems/fa20-lab1-code/blob/master/apps/lab1/lib/intro_lab.ex#L255)
where `%{}` is used to create a new map.

For this project you only need to use two functions from Map:

* [`Map.put`](https://hexdocs.pm/elixir/Map.html) which take a map,
  a key, and a value; and returns a map which is identical to the input except
  that the supplied `key` is associated with the value.
* [`Map.get`](https://hexdocs.pm/elixir/Map.html) which takes a map and
  a key, and returns either the value associated with the key or `nil` if no such
  key is in the map.

  > Now is a good time to talk about arguments that look like `default \\ nil`
  > which appears in the documentation for `Map.get/3`. The `\\ nil` bit here
  > means that this argument is optional, and if it is not supplied the runtime uses
  > `nil` instead.

You should read through the documentation. Knowing more about maps will be useful in
the future.

## Testing

If you want to test just this portion of the lab use
```

> mix test test/kv_store_test.exs

```

If you have been doing things in order, at this point you can use:
```

> mix test

“‘

Which will run all tests. You should ensure that all tests pass before handing in your work.

Handing In

WARNING PLEASE READ THESE INSTRUCTIONS CAREFULLY. YOU MAY **RECEIVE A 0 (ZERO) IF YOU DO NOT**, EVEN IF YOU COMPLETE EVERYTHING THUS FAR.

To handin this assignment:

- First make sure `mix test` shows that you pass all tests. If not be aware that you will loose points.
- Second, make sure you have updated `apps/lab1/README.md`. This requires entering results from Part 2 of the assignment, filling in identifying information, agreeing to the course collaboration policy, and citing your sources.
- Commit and push all your changes.
- Use `git rev-parse --short HEAD` to get a commit hash for your changes.
- Fill out the submission form with all of the information requested.

We will be using information in the submission form to grade your lab, determine late days, etc. It is therefore crucial that you fill this out correctly.