Sergio Alejandro Vargas savargasqu+git@unal.edu.co

An Alternative to Distributed.jl for Pluto

Pluto.jl currently uses the Distributed module that's part of the Julia standard library (*stdlib*) to create worker processes and to send Julia data structures between them. 1 This approach, however, doesn't allow notebooks to use most of the features that make Distributed useful. This project proposes replacing Distributed with a new package that's more specific to Pluto's requirements, with the goal of making Distributed fully available in Pluto notebooks.

Allowing Distributed to be used on Pluto would make Pluto a more complete development environment for Julia, it would make it easier to get started with distributed computing, and it would also open the door for more educational resources on distributed computing to be written in Julia.

The problem

Distributed clearly distinguishes between the initial Julia process (the *master*) and other processes (the *workers*). In particular, only the master process can add or remove worker processes.

Pluto uses Distributed to execute the code of each notebook in separate environments, i.e. the Pluto server is the master process. If a user wants to use Distributed in a Pluto notebook, they won't be able to create new processes because the notebook is executing in a worker process.

While extending the capabilities of Distributed to handle this use case would be ideal, it's rather optimistic to see drastic changes in a module that's part of Julia's *stdlib*, and the problem is limited enough that it could be solved by a smaller package.

Implementation and technical details

The project will be divided into two parts: writing the package and integrating it into Pluto.

The first part of the project consists of the design and implementation of a stand-alone package to create isolated Julia processes. This package should be able to replace `Distributed` on the Pluto codebase, but it will not implement all the functionality of `Distributed`. For the rest of the text this package will be referred to as DA (Distributed Alternative).

DA will consist of a set of functions to spawn separate Julia processes and to send messages to the workers with some mechanism for inter-process communication. Some of the technical considerations DA will need to account are:

Isolation

The main reason Pluto uses distributed processes (as opposed to threads or tasks) is that separate processes can have load separate packages (and different versions of the same package), have separate namespaces, separate memory, and in general are isolated from one another. This is crucial for the correct functionality of Pluto, since isolated environments are necessary for reproducible notebooks without hidden state; one of the goals of Pluto.

Communication

`Distributed` processes communicate by default using TCP (using the `Sockets` module that's in the stdlib). DA could follow a similar design, using TCP sockets and Julia's built-in capabilities for asynchronous programming: tasks and channels.

Other alternatives that could be considered for IPC are:

- Unix domain sockets: Since DA processes will run in the same machine, they could communicate using Unix domain sockets (or named pipes on Windows). This approach could lead to faster communication between processes, but it might be limited by the cross-platform capabilities of Julia (Pluto already has some platform specific issues, see Error Handling below). It will be necessary to study the Libuv documentation (library used by Julia) and to write a prototype to determine the viability of this approach.
- ZeroMQ: Jupyter uses ZeroMQ to communicate between the Jupyter server (master) and the kernels (workers) Julia already has bindings to ZeroMQ, so this alternative could save some time as opposed to implementing the TCP (or the UDS) server from scratch.

Serialization

`Distributed` uses the `Serialization` module (part of the stdlib). Currently there doesn't seem to be any reason not to use `Serialization`, but if one were needed, Pluto already uses MessagePack to communicate with the front-end, and that package could be re-used here.

Error Handling

Pluto is meant to be resilient to errors in the user's code execution (the user should be able to make mistakes!) Worker processes should handle errors appropriately and communicate them to the main process; and the master process should be able to suspend the execution of any worker process.

Pluto has existing problems with signal handling on Windows (see fonsp/Pluto.jl#452),

It may be worth looking into whether signal handling can be properly emulated on Windows via Libuv; or if it's already being done, find why it's not working. In any case, the main goal for DA is providing the functionality that's already works with Distributed, so this issue will only be an optional deliverable, and it'll be addressed if the second part of the project is complete sooner than planned.

For completeness, it's worth mentioning that the features of Distributed that DA *will not* implement are:

- Communication between different machines: All processes will live on the same machine as the Pluto server that spawned them.
- Different network topologies: Pluto only needs to allow communication between the main process and the worker processes (equivalent to `:master_worker` topology of `Distributed`).

Integration with Pluto

The second part of the project consists in integrating DA into the Pluto codebase, replacing all uses of `Distributed`. This includes:

- Porting the actual code.
- Adding a test suite that checks Distributed can be used correctly.
- Updating the documentation for Pluto to make clear how DA is integrated. Since DA will be a bespoke package it's specially important that future contributors to Pluto can understand it with ease.

The project can be considered complete when issue #300 is closed.

Future work

Beyond the main goal of the project, some nice-to-haves might be:

- Resolve issue #452 mentioned in Error Handling.
- Write a Pluto notebook that showcases Distributed or serves as a tutorial on distributed computing.

Schedule

The project can follow the regular GSoC timeline, starting in June 6 and ending in September 12. I may be unable to code during the penultimate week of the program (first week of september). However, at that point most of the coding tasks should be complete.

The breakdown of the project for the 12 weeks of the program might look like:

- W1: TCP and UDS server prototypes
- W2: Study ZeroMQ
- W3: ZeroMQ server prototype

- W4: Settle on an IPC implementation.
- W5: Buffer (in case of any delay)
- W6: Finish package and present as part of the first evaluation
- W7: Familiarize with Pluto code base
- W8-10: Integrate package with Pluto
- W11-12: Tests and documentation