Università di Pisa

Distributed Systems and Middleware Technologies:
Fedlang Framework: Porting to the Go language

Fabio Piras, Giacomo Volpi, Guillaume Quint

Academic Year: 2023/2024

Contents

Chapter 1

Introduction

Fedlang is a research project of the University of Pisa, which aims to provide a platform for federated learning experiments: the middleware of Fedlang is written in Erlang and client and server components are written in Python. The goal of this project is to port the existing client and server components of Fedlang in Golang, particularly focusing on the federated c-means algorithm to prove that the Go language can be used as an alternative to Python for the development of a federated learning platform. We also extended the existing framework functionalities to allow for a generic network topology, implementing a ring configuration as an example.

All of the code developed is available over GitHub at the following link: `https://github.com/glmquint/fedlang` and it will be referenced throughout this document.

1 Previous work

The pre-existing work can be found at the following GitHub repository `https://github.com/jlcorcuera/fedlang` and it consists in a study of most of the aspects related to Machine Learning and Deep Learning, oriented in deciding the right enhancements to apply to Fed-Lang, an Erlang and Python-based framework for supporting data scientists in the development of Federated Learning algorithms. The main focus of the project, with respect to other frameworks is the presence of an Erlang middleware for handling the communication aspects, and the possibility to support any kind of Machine Learning and Deep Learning Python framework. There were already some implementations of the federated learning algorithms, namely FedAvg, Federated Fuzzy c-means and Federated TSK-FRBS.

2 Initial study

We started the project by studying the existing codebase of Fedlang, to understand the architecture and the functionalities of the system; for this purpose a Docker image was built to reproduce the environment of the system and to avoid any compatibility issues. We then ran multiple tests to ensure that the system was working correctly and to understand how the middleware interacts with the client and the server. We then studied the Go language, to understand how to implement the functionalities of the system in Go, and to understand how to communicate with the Erlang middleware. Before venturing any further, we need to understand why we chose Go as the language for the porting of the system and what are the main differences between Go and Python.

2 .1 Why Go?

Go is a statically typed, compiled language that is designed to be simple and efficient. It was created by Google in 2007 and it is used in many Google projects, such as Kubernetes, Docker, and others. Go is designed to be simple and efficient, with a focus on performance and scalability. It is a compiled language, which means that it is faster than interpreted languages like Python. Go is also statically typed, which means that it is more reliable and easier to maintain than dynamically typed languages like Python. Go has a garbage collector, which means that it is easier to write memory-safe code in Go

than in languages like C or C++. Go has a built-in concurrency model, which makes it easy to write concurrent programs in Go. Go is also designed to be easy to learn and use, with a simple syntax and a small number of keywords.

3 Development process

The first main problem was to find a library in Go that could replace PyErlang, which is used in the original system to communicate with the middleware; we found Ergo framework [1], which allows us to communicate with an Erlang node using the Erlang distribution protocol. For this reason, we implemented a simple ping-pong test in order to have a proof of concept of the communication between the Go client and the Erlang middleware, to ensure that we can achieve the same functionalities of the original system. After that, we started to port the server component of Fedlang in Go, also implementing the fedlang_process class.

At this point we had an hybrid system, where the server was written in Go and the client in Python (the code can be found as version 0.1[2] of the repository cited at the beginning of the document). This proof of concept allowed us to understand the feasibility of the project and to understand the difficulties that we could encounter during the development of the system, but it also allowed us to understand the potential of the Go language in the development of a federated learning platform.

We then completed the porting of the client in Go (v 0.2[3]): this allowed us to change the serialization protocol and to finally tweak both client and server with goroutines and go channels. This allowed us to have a fully functional system in Go . One prime example of the benefits of the newer language is the serialization protocol: in the hybrid system, the serialization was done using a library implementing the Pickle protocol[4], which, although very powerful and flexible, is also slower than the standard Go serialization library. In the full Go version of the system, we used the gob and json libraries for serialization, which are widely used, faster and well suited for the task.

Finally, we tested the system to ensure that the porting was successful and that the system was working correctly, for this scenario we used a docker network and multiple containers for each service to mimic a fully distributed environment. The results of the tests were positive and we were able to demonstrate that the Go language can be used as an alternative to Python for the development of a federated learning platform.

4 API enrichment

We added a feature to the system that allows nodes to communicate with each other: this was done with the purpose of leaving to the developer the possibility to use a different topology for the communication between nodes.

[1] `https://github.com/ergo-services/ergo`
[2] `https://github.com/glmquint/fedlang/releases/tag/v0.1.0`
[3] `https://github.com/glmquint/fedlang/releases/tag/v0.2.0`
[4] `https://github.com/MacIt/pickle`

Chapter 2

Code organization

The code we developed is organized as follows:

- **FedlangProcess.go** contains core functionalities both for the client and the server. It provides support for all communications between nodes and middleware.
- **fcmeans_client.go** contains the client component of Fedlang, which is responsible for executing the steps of the federated learning algoritm.
- **fcmeans_server.go** contains the server component of Fedlang, which is responsible for aggregating the results coming from the clients.

1 FedlangProcess

A specialization of the `FedlangProcess` struct must be instantiated using the generic method

```
func StartProcess[T any](
 go_node_id,
 erl_cookie,
 erl_client_name,
 erl_worker_mailbox,
 experiment_id string)
```

where:

- `go_node_id` is the identifier of the Go node.
- `erl_cookie` is the Erlang cookie.
- `erl_client_name` is the name of the Erlang worker node.
- `erl_worker_mailbox` is the name of the Erlang worker mailbox.
- `experiment_id` is the identifier of the experiment.
- `T` is the type of the struct containing information about the server or the client of the distributed algorithm

Both Client and Server must implement some methods that will be called by the middleware. Those methods should be exported by the package developed by the algorithm programmer (i.e. make them Capitalized). This allows the FedlangProcess to view and call them.

All methods called by the FedlangProcess can have a variading number of parameters and always contain as the last parameter a copy of the FedlangProcess that called them. This is useful if the programmer wants to use some advanced functionality offered by the FedlangProcess, like a peer to peer communication between nodes during the experiment execution.

If a method that is invoked by the FedlangProcess returns an `etf.Term` object, the FedlangProcess will send it to the Erlang node that called the method. This allows for a more opaque interaction between the distributed algorithm and the middleware, as it doesn't require the developer to know about any communication detail.

2 Client

The client must implement the following methods:

- `Init_client` initializes the client, taking an encoded json string as configuration.
- `Process_client` executes one step of the federated learning algorithm. Takes as input the experiment id string, the current round number and the encoding of the current centers.
- `Destroy` is called when the experiment is over. It can be used to free resources. Generally kills the computing node.

3 Server

The server must implement the following methods:

- `Init_server` initializes the server, taking an encoded json string as configuration.
- `Process_server` aggregates the results coming from the clients. Takes as input the experiment id string, the current round number and the encoding of the results.
- `Destroy` is called when the experiment is over. It can be used to free resources. Generally kills the computing node.

4 Erlang

The already existent Erlang code has been modified to allow the use of different programming languages for distributed algorithms development. Many hardcoded variables and functions now accept the `CodeLanguage` atom that is passed as an argument to switch only the relevant piece of code necessary for the different implementations. Other than that, the only improvements to the middleware API are the addition of a new message `fl_worker_result_ack` that can be returned by the worker instead of the classic `fl_worker_result`. This new message allows the `StrategyServer` to keep track of the workers that correctly elaborated their result but didn't send them immediately. This in turns allows the algorithm developer to defer the sending of the results to the server to a later time, for example after the aggregation of all the results in a peer to peer fashion.

Chapter 3

Different Topologies

The original middleware implementation assumed a strict separation of roles between nodes. Each worker would execute independetly and communicate only with the director node, which in turn would aggregate all responses coming from workers.

Even if this solution is simple to implement, it leads to synchronized moments where the communication bandwidth is entirely occupied by results coming from all clients (with each usually containing large amount of computed data). To reduce this phenomenon, which worsen with the number of connected clients, we would like for the workers to perform part of the aggregation, easing the workload for the server.

Distributed algorithm developers might then need for a custom topology for organizing peer to peer communication.

In version `v0.3.0`[1] we developed an alternative federated fuzzy c-means clustering algorithm which uses a ring topology for a distributed aggregation of locally computed results before sending them to the central server. At each round, a client is selected in a round-robin manner as the one aggregating other nodes' data before sending the final result. All other nodes simply relay their computed results to the next node in the ring.

To allow peer to peer communication, the `FedlangProcess` implements a `PeerSend` method with the following signature

```
func (s *FedLangProcess) _peerSend(
 dest_selector func(id, num_peers int) int,
 msg etf.Tuple
)
```

The first argument is a function which will be called to select the next node in the ring, given the current node id and the total number of peers in the network. The second argument is the actual message that needs to be sent.

The information of how many nodes are in the network shouldn't be considered a constant. The number of nodes can change during the execution of the algorithm, as nodes can join or leave the network at any time. To mantain the ring topology, therefore, the programmer should describe the next node in the ring as a parametric function of just these two unkowns: the current node id and the total number of nodes in the network

The update of the network topology is managed by the middleware, which will call the `Update_graph` method with the updated number of nodes in the network. To mantain backward compatibility with the existing python algorithms implementation, the update network event is non-blocking, i.e. it doesn't require an aknowledgement from the clients.

The `PeerSend` method doesn't return the result of the operation. If the message cannot be sent (for example, if we requested to send a message before any network update has been managed), the message gets cached by the `FedlangProcess` and a new sending attempt will be made after all subsequent network update events.

[1] `https://github.com/glmquint/fedlang/releases/tag/v0.3.0`

Chapter 4

Conclusion

1 Performances

After completing the translation of the fuzzy c-means algorithm into Go, we investigated the overall perceived slowdown of the application.

Using the standard Go profiler, pprof, and the CLI tool `go tool pprof`, we observed that both the server and client components were slowed down by the 'number crunching' operations required by the clustering algorithm.

Although the final project efficiently utilizes goroutines to maximize the use of available hardware, it was necessary to manually implement many arithmetic operations that are otherwise already provided by specialized libraries in Python. We believe this to be the primary source of the slowdown, in addition to the use of the Ergo framework for communication with the Erlang middleware, which has less support compared to the more widely known Pyerlang.

This is reflected by the fact that Python is more suitable for tasks related to machine learning, given its long history of optimized libraries specifically designed to support this domain.

2 Future Work

The project of porting the Fedlang framework from Python to Go has proven to be a valuable endeavor, not only in achieving the primary objective of demonstrating Go's viability for federated learning systems but also in expanding the framework's capabilities.

Throughout the development process, the challenges of finding suitable libraries and ensuring seamless communication between the Go components and the existing Erlang middleware were successfully addressed. The use of the Ergo framework and the development of hybrid systems allowed for a smooth transition and ensured that the core functionalities of the original system were preserved and enhanced.

The introduction of a ring topology for distributed aggregation represents a viable alternative over the previous architecture. It not only reduces the communication bottleneck that can occur in traditional client-server setups but also opens up new possibilities for distributed algorithm developers to implement custom topologies that better suit their needs.

In conclusion, future work should focus on further enhancing the libraries supporting machine learning operations and improving communication performance within federated learning environments. These advancements would significantly bolster Go's potential in this domain, enabling the development of more efficient and scalable systems. Additionally, fostering greater support and optimization for frameworks like Ergo could contribute to smoother integration and more robust communication in hybrid architectures. Such efforts will be crucial in solidifying Go's place in the evolving landscape of machine learning and distributed computing.