Gomat specification

Gauthier Jolly Kévin Carénou Matei Oltean

10th January 2018

Contents

1 Introduction

`Gomat` is a tool which allows users to compute complex operation on matrices on a computer network. `Gomat` is in the same time a `Go API` and a daemon. By running the daemon, a user becomes part of a network and can use the `Go API` to execute big computations.

2 Goal and Functionalities

2.1 Daemon

When a user wants to join a `Gomat` network, it has to run the dedicated daemon. The daemon is the main component of the application. It splits and merges, executes computations and communicates with other peers. For this last point, it uses a `Gossiper` network.

2.1.1 GUI

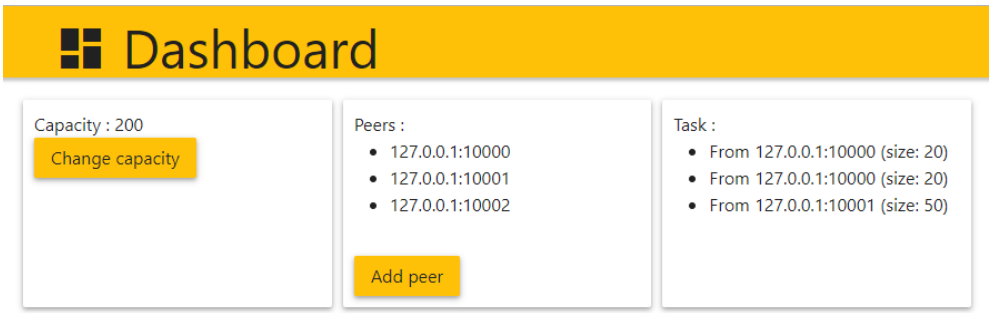

Figure 1: Global view

On the `GUI`, the client can change the maximum computational power it brings to the network through the daemon. He can see the tasks he is currently computing, the source of the computation and the size of the computation, check the peers he knows and add some if needed.

2.2 API

Our API is a `Go` package which is used to communicate with the daemon. A user that wants to compute an operation between to big matrices calls

```
Gomat.compute(matrix1, matrix2 Matrix, operation Gomat.Operation)
(result Matrix, err error)
```

2.3 Computations redistribution and failure management

We want to be able to redistribute properly the tasks, from the master node to the others, be reliable and fast, despite the networks flaws.

Since it is a decentralised system where one peer delegates part of its job to others, it is essential to detect failures, manage them, and implement methods that make the system more resilient.

It must be noted that failure detection and management work both ways. For a given computation, the gossiper that wants its job done must detect failing peers, so that it does not have to wait forever for their reply.

3 Design and Architecture

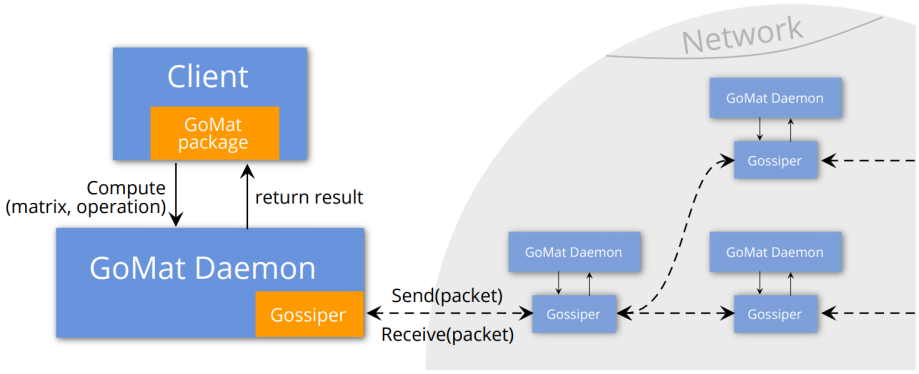

Figure 2: Global view

3.1 Daemon

The creation of GoMatCore package and the demonstration application is supervised by Kévin Carenou.

A `Gomat` peer is not a `Gossiper` peer, but contains a `Gossiper` peer that it uses to send and receive messages. Our `Gomat` application uses the `Gossiper` as a transport layer (layer 4).

3.1.1 GUI

Once the daemon is running, it is not automatically connected to a `Gomat` network. The user has to open a web browser and go on `http://localhost:`

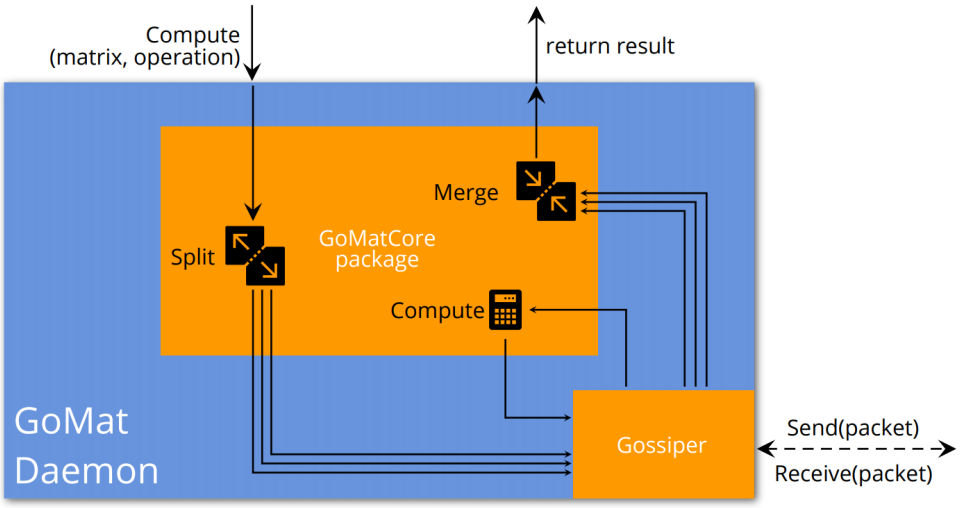

Figure 3: Daemon architecture

`8088` (can be modified in a configuration file) to add some peers to be connected on a `Gomat` network.

3.1.2 Split, merge and computation

When receiving a computation request from the API, the daemon splits the input matrices into smaller submatrices, to be sent to the network for computations. This submatrices are identified by their position (row/column) as a block of the original/result matrix.

When a peer's deamon accept to compute a chunk of data, this deamon will do the computation using the `GoMatCore` package. This package implements functions computing the different operations.

After receiving the results of the computations, the daemon merges the subresults into a result matrix. Some operations such as multiplication needs several steps to compute. In the case of multiplication, we first need to compute the multiplication between submatrices and we get the results. Then we launch the addition of block to get the result matrix.

3.2 API

The API in charge of the communication between the application and the Daemon will be implemented by Gauthier Jolly.

Our API is a `Go` package which is used to communicate with the daemon. When an application wants to compute an operation between to big matrices,

he calls:
`Gomat.compute(matrix1, matrix2 Matrix, operation Gomat.Operation) (result Matrix, err error)`

This function is blocking.

If there is no `Gomat` daemon running on the computer, `Gomat.compute` returns an error.

3.2.1 Communications between API and Daemon

The application using the API and the Daemon are running on the same computer. Thus, there is no need to use the network (i.e. the loopback interface). We use `Inter-Process Communication (IPC) sockets` to send information between the daemon and the API.

3.3 Computations redistribution and failure management

Computations redistribution and failure management is supervised by Matei Oltean.

3.3.1 Computations redistribution

Each gossiper doing computations has a maximum computational power, and a current one (reduced for every current computation).

When a gossiper receives a computation order from the API, the following happen:

- That gossiper splits the matrices, which should be of a maximum size `self.MaximumComputationalPower`. There is a trade-off, since if the matrices are too small, there is no real gain (too much time is lost splitting and sending), and if they are too big, nobody will be able to take care of the computation. So, we decided that only the initial gossiper can split the matrices, and that they should be moderately big.

- Then it sends pairs of submatrices, with the operation to perform on them (as subtasks) to random alive peers or itself. They can either accept the task, or not. For each task, after a timeout, the task sent is not being taken care of, it is sent again to another random peer.

- When a peer receives a subtasks, it checks if it can compute it. If it is too big, it send it again to another random peer. To ensure that these

messages do not keep being sent indefinitely, they have a `HopLimit` field, which is decreased every time it is sent again, and the message is discarded when it reaches 0. If the peer can take care of the computation, it does not start it yet, but sends a message to inform the initial node that it is ready to do so, and waits for an acknowledgement to start.

- For each subtask, the master node selects the first node that can take care of it as the owner of the computation and sends it a message back to inform it.
- Then, it waits for all the subresults, and when every computation is done, the submatrices are merged and sent to the API.

3.3.2 Detecting failures

To manage failures, we first need to detect them. No failure detector can be perfect, but it must be eventually perfect. We also need to make sure that we do not saturate the server with ping messages to detect failures, since that could lead to other failures. Also, we do not want to suspect other nodes too early, since it is costlier to have less peers than having detected failing ones too late.

We used a heuristic similar to the *route flap damping* of BGP. Whenever we add a peer, we set a counter for it, starting at 0. Every *timer* seconds, we send it a message and we increment the counter. Whenever we receive a status message from him, we decrement it. A peer that has a counter above a certain (small) threshold is considered to have been failed.

3.3.3 Failure management

When a gossiper g finds that another one, a, has failed, it will be removed from the list of peers, the routine created to communicate with a will be killed and one of the three will occur:

- If a was doing some computation for g, g will split the computation between its remaining peers.
- If g was doing some computation for a, a will stop its computation, and not send the result back anymore.
- In any other case, nothing more will be done.

3.3.4 Making the system more resilient

The master node will add gossipers to its list of known peers, and the network will be more robust.

Also, since the failure detector is not perfect, there are timeouts on every routine involving other peers. Upon a short timeout, the task will be dispatched to another peer.

4 Evaluation Plan

First, we needed to be sure, that the matrix splitting, merging, and the operations on them behave as they should.

To test the implementation of these operations, unit tests using the `testing` package available in golang has been written and launched after every change of any of the operations. We also launched tests about splitting, computing on submatrices and then merging the blocks. This mimics the behavior of the final system, the only difference is that the data is not sent through packets.

Since most of the tests will be run through the API, we also checked that when `Gomat.compute` is executed, the correct tasks are run in the background.

Since the message spreading is built on top of Peerster, it should work and is already tested. Obviously, the messages sent for computations and failure detection are different from GossipPackets, but we checked if everything works when testing our parts of the program.

For the failure detection, we checked two things (and also adapt the timeout parameters accordingly): that if a peer fails, the other detect it fast, and that there a very few false positives (i.e. that a node is seen as having failed, while it still runs fine). We simulated small networks, with failing nodes (i.e. deconnexions), and checked the reactions of the remaining gossipers.

We also checked that when a node receives a message, it always adds the sender to its list of known peers, by sending blank messages (we must also be sure that a message not well formatted is discarded, instead of making the node thread crash).

While all those tests are running, we have a GUI running, and we are checking that all the informations displayed are correct.

5 Conclusion

The system has become a little bit more centralised—originally we wanted every gossiper to be able to split the input matrices, not only the original one. But, there is a trade-off between decentralisation here, because of the merging and the network.