

# Gomat specification

Gauthier Jolly      K vin Car nou      Matei Oltean

10th January 2018

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Goal and Functionalities</b>	<b>2</b>
2.1	Daemon . . . . .	2
2.1.1	GUI . . . . .	2
2.2	API . . . . .	2
2.3	Computations redistribution and failure management . . . . .	2
<b>3</b>	<b>Design and Architecture</b>	<b>3</b>
3.1	Daemon . . . . .	3
3.1.1	GUI . . . . .	4
3.1.2	Split, merge and computation . . . . .	4
3.2	API . . . . .	4
3.2.1	Communications between API and Daemon . . . . .	4
3.3	Computations redistribution and failure management . . . . .	5
3.3.1	Computations redistribution . . . . .	5
3.3.2	Detecting failures . . . . .	6
3.3.3	Failure management . . . . .	6
3.3.4	Making the system more resilient . . . . .	6
<b>4</b>	<b>Evaluation Plan</b>	<b>7</b>
<b>5</b>	<b>Conclusion</b>	<b>7</b>

# 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

On the GUI, the client can change the maximum number of computations his computer can do at a time. He can see the number of computation his computer has done, check the peers he knows, see the ratio  $r = \text{number of computation tasks sent} / \text{number of computation tasks computed}$ .

### 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 two 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 gossipier 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

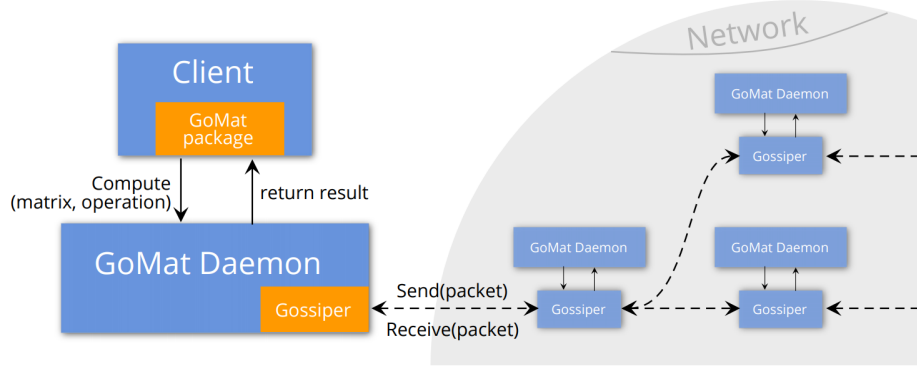


Figure 1: Global view

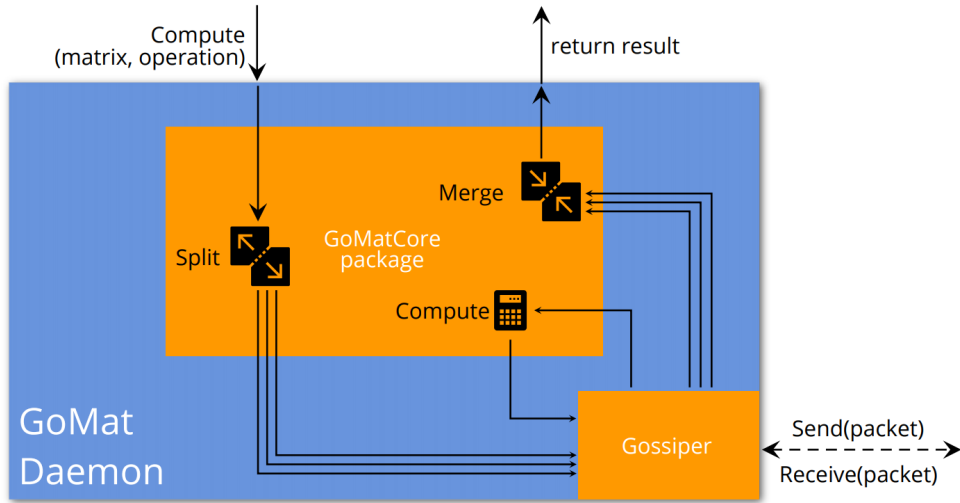


Figure 2: Daemon architecture

#### 3.1 Daemon

The creation of GoMatCore package and the demonstration application will be supervised by K vin 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:8080` (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 will split the input matrices into smaller submatrices, to be sent to the network for computations.

When a peer's daemon accept to compute a chunk of data, this daemon 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.

## 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 two 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 will be supervised by Matei Oltean.

#### 3.3.1 Computations redistribution

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

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

- That gossipier 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 gossipier 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 gossipier  $g$  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

If a gossipier  $g$  receives a task and has enough idle peers, it can redistribute some of it amongst them. There must be some wait time (maybe they have all received subtasks from the same job, but because of delays,  $g$  still sees them as idle). In that case, the ‘master’ peer must be informed of their computations, they will be added to its peers, and as for others, it must now detect their failures.

Then, if  $g$  crashes, not all of the computations sent to him will have to be redone, since some subtasks are now processed by its peers.

The master node will also be able to 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 need 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 will be written and launched after every change of any of the operations. Also, after being sure that they work locally, we will also test them within a small network.

Since most of the tests will be run through the API, we must also check 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 will check if everything works when testing our parts of the program.

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

We will also check 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 will be running, we will have a GUI running, and we will check that all the informations displayed are correct.

## 5 Conclusion

These specifications are meant to change, since it is only a second draft.

For example, a function of preference should be added. The sizes of the matrices sent for computations should depend on the computation power of the gossipier, its response time, etc. But for that, we would also need to optimise the splitting and merging parts, in order to be able to efficiently merge submatrices of different sizes.