Misra for termination of computation

The Misra algorithm

Misra's algorithm (named after Jayadev Misra) allows for detection of termination of computation in an arbitrary network. This problem is qualified in the original paper as of "considerable importance".
Misra's algorithm uses a marker that travels through the network. The marker carries information and modifies it depending on the internal state of the node where it currently is. Once a threshold is reached in the information carried by the marker the computation is considered terminated.

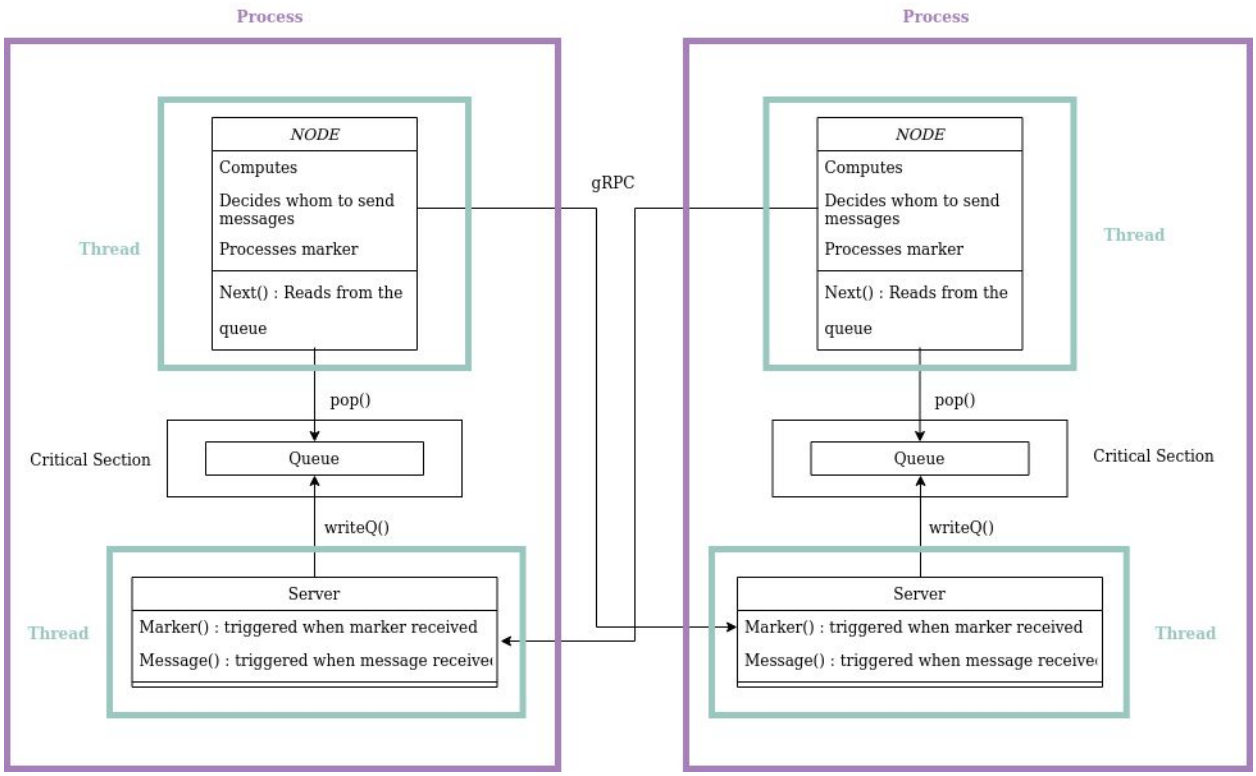
We conceptualize the network as a graph where processes are vertices and communications links are edges. If every process in the network is idle and there are no messages in transit, then computation has finished. If the marker traveled through every possible edge in the network, while only encountering white processes, then computation has finished. This conclusion is based on the assumption that the marker is propagated through the same communication channel as the messages triggering computation in processes, and that all messages are received in the order in which they have been sent.

The marker knows whether or not it traveled through all the edges by incrementing a number. Once that number reaches the length of the cycle which includes every edge of the network, then we deduce that computation has finished. This number is reset when the marker encounters a process that has been active since its last visit. This property is modeled by colors: black and white.
Once a process receives a message, it paints itself black.
Once a marker leaves a process, it paints the process white.
Thus the number is reset when encountering a black process.
To detect whether or not a marker is going through an unvisited edge, we use the depth first search algorithm.

Our implementation

High level architecture of the system

In our implementation we used **python** and **gRPC** for the communication between the nodes. Here is a diagram of the system:



As you can see, the node class implements the simulated computation (which basically slows time), which results in a random choice of destination addresses for the sending of the messages. In the node is also processed the marker; the entirety of the misra algorithm is implemented there. The node maintains an internal state to allow for the DFS and the misra algorithm to operate. Finally, the node can send marker or messages through gRPC.

Once a node finishes what it has been doing (processing the marker, computing, choosing destinations and sending…) it fetches from a queue its next instruction.

That queue is protected by a critical section and is populated by requests by another thread that runs a gRPC server implementation.

That gRPC server receives itself requests by other nodes objects through gRPC.

Thus, each node has its own server, which runs in another thread and its own queue of messages/markers. All of them of them live in the same process. Communications between nodes, thus between processes is achieved through gRPC. **No shared memory is used**.

The Node Class

In the node class we find various variables dedicated to diverse functions. Some are used to generate a controlled amount of randomness to simulate the behavior of a real distributed process (sending messages to particular nodes), some are for the modalities of communications between nodes (addresses to the other nodes, own address) and some are used to carry out the misra algorithm along with the DFS.

We store in this class: the number of unvisited edges the marker has traveled through, the color of the process, the number of rounds “leaved” by the last encounter with the marker, the length of a cycle in this particular network, and various arrays such as the array of addresses on which the node hasn’t yet sent the marker, the fathers and sons arrays for the DFS.

The informations that are sent from node to node along with the marker are: the address of the sender, the number of rounds at which the marker is at, and the number of unique visited links for this round.

In the case where the node receives a message, we simulate its computation time using a sleep function, and we then send to a randomly chosen amount of randomly chosen nodes a message that will trigger computation on them. As the execution of the system progresses, a node has less and less chance to send new messages to other nodes. This simple simulation is designed to demonstrate the capabilities of our implementation.

User Guide

By default, we assume that the network is a fully connected network of 4 nodes. The amounts of nodes can easily be changed by adding or deleting elements on the table array at the beggining of the code. This array stores the addresses of all the nodes.
To create an arbitrary graph, one could make as many copies of the table as there are nodes, and modify each table (each “belonging” to one node) to represent the edges of each node. Theoretically, the algorithm should work on these, but we haven’t implemented a way to make an arbitrary graph in an easy manner.

To run the system with the default example (4 nodes), we need to open 4 terminals (one for each node and its output), and issue on each terminal these commands :
python3 misra.py 1 1
python3 misra.py 2 2
python3 misra.py 3 2
python3 misra.py 4 2

The first argument defines the address of the node started by the terminal such that :
Address of node = "localhost:5005<arg1>"

The second argument defines whether this node is an “initiator” (arg2 = 1) or not (arg2 = 2). What we call initiator is the node which will receive the first message of request of computation, as well as the marker (both messages are appended to the queue, in order).

The last step is to trigger the initiator by typing the character ‘1’ in its console (we do that because we wait for the user’s signal since he must have the time to setup and issue the commands in all terminals).

In order to interpret the messages appearing on each console:
(nb = number of unique edges visited this round)

<sender> ==== <nb> : the node received a marker with that nb from sender
<nb> ==== <destination> : the node sent a marker with this nb to that destination
MARKER RESET : nb has been set to 0 and the round number has been increased because the marker has stumbled upon a black process