Multi-Agent System for Fault Detection in Distributed Systems

Joana Coutinho
87666
joana.coutinho@tecnico.ulisboa.pt

Rafael Soares
87675
joao.rafael.pinto.soares@tecnico.ulisboa.pt

Pedro Rodrigues
87696
pedro.p.rodrigues@tecnico.ulisboa.pt

1 PROBLEM DESCRIPTION

In Distributed Systems, one of the most prominent problems is figuring out if a server is acting correctly. A correct server is one that respects its specification, meaning answering correctly to requests and in a timely manner. Knowing if a server is correct is of extreme importance for most Distributed Systems architectures, as they might be needing answers from servers which are actually crashed.

At first glance, this problem seems trivial to solve: a server either answers to a request or it does not. However, in real distributed systems, there are many problems to this:

- The server may be busy, not being able to respond on time;
- The network can be overloaded, with the request being lost or delayed;
- The server may be infected or malicious, hiding its malfunctions.

In order to answer this need, modules are added called Fault Detectors [1], which job is to conclude whether or not a server is correct or not by making queries to said server. They must be able to adapt to the environment in order to give the most accurate state of the system as they possibly can and communicate the state to other Fault Detectors.

However, just as servers can be malicious, so can Fault Detectors, being able to lie and deceive others about the correctness of themselves and the system.

1.1 Geo Replicated Systems and Bitcoin Malware

Let us imagine a Geo Replicated system, where different replicas of our server are in different geographic locations in order to provide lower latency to clients around the globe [5].

Since the rise in popularity of Bitcoin [3] and crypto-currencies as a whole, there has been an increase of attacks that infect servers and computers alike in order to mine these currencies, hiding themselves while keeping appearances of normal working [4]. This infections can be spread over the whole system, taking a significant toll on our system performance.

As such, we need a way to observer changes in server response latency's in order to identify and extract these servers from the system to stop the spread of infection.

Given these problems properties, we could imagine a new take on the Fault Detection problem by using a Multi-Agent approach, as these are generally decentralized and able to adapt for better decision making on server correctness.

April 2020, 2020

2 MULTI-AGENT FAULT DETECTION

Agents working as Fault Detectors should be able to detect faulty servers, communicate its state to other agents and question the state of others, as these can be infected. With a multi-agent implementation, correct Fault Detectors can cooperate with each other to decide whether or not a certain Fault Detector is telling the truth about its state or if it has been infected, while infected Fault Detectors will cooperate between themselves to infect the whole system.

The goal of the malicious Agents is to spread their infection and become the majority in the system.

The goal of the correct Agents is to identify malicious entities and remove them from the system for repairing.

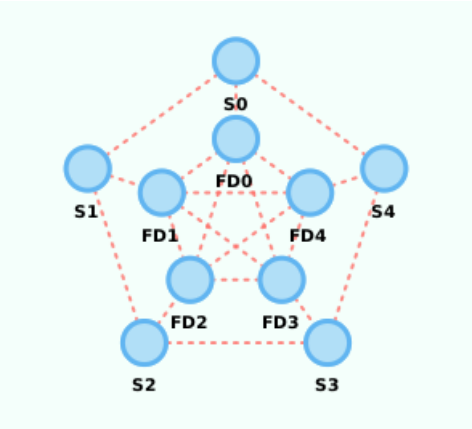

Figure 1: Visual representation of the system

2.1 Functional Requirements

- Each Fault Detector must be able to adapt to the fluctuation of the network and workload of the server to give the most accurate state of the system;
- Each Fault Detector has N neighbours of which it is responsible for detecting whether they are malicious or not;
- When a Fault Detector believes a pair Fault Detector/server should be removed from the system, it requests a quorum centralized in the suspected pair where the majority vote wins;
- Malicious Fault Detectors want to remove healthy Fault Detectors and keep its Infected Network in the system for as long as possible;

2.2 Assumptions

- Inside Infection occurs when an entity receives a message from an infected entity, with a certain probability;

- Once the system is infected by an outside source, it is only infected again once the infection is successfully removed;
- An infected server will always answer a request;

2.3 Communication

To represent the communication in our system, we will use a network simulator to simulate the communication between entities.

When a server or fault detector wants to send a message, they submit it to the network simulator with the destination and content of said message. Then, the network simulator calculates the distance delay and once the message is ready to be delivered, it calls a method of the destination that processes the message.
Next, we have the interactions between the entities of our system:

2.3.1 Module I - Server <-> Fault Detector. A Fault Detector pings every server in the network. The server answers within a certain time period $t \in \{t_0, t_1\}$, which depends on the server workload. When the server is infected, the period is increased by a variable that represents the extra work load (for example, the server is being used to mine crypto-currency).

Although a Fault Detector pings all servers in the network, it is only responsible for its neighbouring ones, due to the unreliability of pings of far away servers. Nevertheless, information is needed on other servers, as we will see next.

2.3.2 Module II - Fault Detector <-> Fault Detector. When a Fault Detector perceives that one of its neighbouring servers has been infected, it requests a quorum consensus [2] for said server in order to decide whether the pair should be removed or not. This Quorum is centralized in the server, meaning the participants of the quorum will be the K nearest neighbours on each side of the server.

When a Fault Detector receives a quorum request, they check whether they agree that the server must be removed and send the response to every Fault Detector in the network. Thus, eventually every agent will reach a consensus and remove or maintain the server their network.

In the case of removal, we also remove the Fault Detector closer to it in order to maintain the ratio of Fault Detectors to servers.

When a server is removed, we need to reconstruct our network structure. We close the gap left by the removed server/Fault Detector by connecting each of its closest neighbours together. After some time, the server comes back online in a healthy state, being added to the network again in the same place.

2.3.3 Module III - Server <-> Server. In order to represent a real network, each server sends a work request to its neighbours and responds to work requests by other servers.

2.4 Infectious Behaviour

Once an entity has been infected, its behaviour will change to a malicious one. The entity registers itself in a Infected Network, in order for every infected entity to know of each other.

2.4.1 Infected server. First, a server gets infected by an outside source. An infected server will now simulate that it is doing some form of background work, adding a constant delay D to its responses. Every future message sent by the server will hold a chance to infect another entity, spreading the infection throughout the system. Once a server has been infected, it will stop making work requests to other servers, but it will still answer requests from others, as said in Module III.

2.4.2 Infected Fault Detector. Eventually, a Fault Detector gets infected by an infected server. The infected Fault Detectors will now want to keep their infected servers in the network the longest possible in order to infect more of the network, while trying to remove correct servers from the system to gain more influence.

When a Quorum is called, the infected Fault Detector bases his decision on two factors:

- If the quorum's target is a healthy server/Fault detector pair, it will always vote to remove them;
- If the quorum's target is an infected server and/or Fault Detector (since removing a server means also removing its closest Fault Detector, which may be infected), it will always vote not to remove them.

3 SYSTEM PROPERTIES

3.1 Agent Properties

The agents in our system have the following characteristics:

- **Autonomous**: the agents learn depending on the environment and make their own decisions accordingly;
- **Adaptive**: the agent adapts to the level of communication variance that exists in the network, learning this from the experience of communicating with its server;
- **Rational**: it is able to act maximising its ability to detect infected servers/fault detectors while minimising the wrongly considered infected;
- **Not curious**: the agent doesn't look for new ways to solve the problem, it uses a predefined strategy to achieve it's goal;
- **Reactive**: it reacts to the fluctuation of network delay, taking it into account when deciding whether or not a server has been infected or has crashed;
- **Proactive**: the agent takes initiative by communicating with other agents whenever it thinks a neighbouring server or fault detector is infected;
- **Social**: the agents interact with each other to decide if a particular server is infected or not;
- **Collaborative**: the agents that have not been infected collaborate to identify servers/other agents who have;
- **Not believable**: it is not believable, since it does not have an avatar and it does not have human-like behaviours;
- **Not mobile**: the agents are always in the same place and attached to the same server, so they are not mobile;
- **Have a personality**: the agents can act differently due to the environment and/or due to being infected. For example, an agent in a low latency environment will consider a server infected after less time than one in a higher latency one. Also, the agents will have varying trust levels. For example, if an agent's server is infected, if the agent is more cautious it will never trust it's neighbours, whereas if it's more trusting it will give them the benefit of the doubt;
- **No veracity**: if an agent is infected it can knowingly communicate false information.

3.2 Environment Properties

The environment has the following properties:

- **Inaccessible**: The information that is perceived can be inaccurate since an agent can lie.
- **Non-deterministic**: servers that are infected will try to infect others non-deterministically.
- **Dynamic**: As it is a multi-agent system, their decisions are in parallel which means that an entity may be infected while others are deciding.
- **Discrete**: There is a finite number of possible actions and perceptions such as to trust or not to trust, to be healthy or to be infected, to ping or not to ping, to request a quorum or not.
- **Non-episodic**: The trust between the agents is only reset after an agent is removed, being recalculated every tick.

3.3 Agent interaction with the environment

The environment is the one to initially infect the system (Outside Infection), at a random time and targeting a random server. While the system is dealing with an ongoing infection, the environment doesn't infect it again.

An agent interacts with the environment by using the network simulator to send and receive messages from other agents and servers. This network simulator can also carry an infection to new entities in its messages (Inside Infection).

4 TYPES OF AGENTS

For the study of the system, we prepared different agents with different techniques in order to try and identify one that could correctly and efficiently identify infected entities, surviving the longest time possible in the system.
The success metric is to remove infected from the system before having a majority of consecutive infected fault detectors in a quorum, since if, for example, the quorum size is 5, 3 consecutive infected agents would always be able to remove the healthy ones.

4.1 Baseline Agent

Our first agent is the Baseline Agent. This agent has two different trusts: the trust of each server and the trust of each Fault Detector. For the first type of trust, the agent pings all servers at a constant rate, learning it using a normal distribution. For the latter, the agent takes into account the responses of the quorum such that if a Fault Detector disagrees with it, then its trust lowers, but if it agrees, then the trust grows.

4.2 Memory Agent

The Memory Agent also has two trusts like the one before, but differs from the baseline in the calculation of server trust. For this agent, we wanted to see how the last X pings differ from all of the previous pings in order to detect sudden changes in the ping responses. For example, once a server is infected, instead of slowly changing the normal distribution, we have the last X pings and, since the server is infected, the average of said pings should become significantly higher than our normal distribution. Since adding points from an infected server deviates the normal distribution, in this agent, we only add a point once X ticks have passed, and only if the server is not suspected. If it is suspected and removed, we clear the last X pings so it doesn't affect the normal distribution.

4.3 Memory and Ping Learning Agent

The Memory and Ping Learning Agent continues from the previous agent by adding the improvement of learning the ping of each server. This allows an agent to send pings to each server at different rates depending on the distance, hopefully allowing it to detect faster that a neighbour server was infected.

4.4 Perfect Agent

We implemented a Perfect Agent, that knows the server properties, such as the minimum and maximum time to answer and the answer delay when infected. The purpose of this agent is to analyse the best accuracy and time for detection we could possibly have. Since this agent requires perfect knowledge of the system properties, it is not realistic. However, it is useful to gauge how close agents with no knowledge of the system can come to these values.

5 ANALYSIS OF THE AGENTS

To analyse the performance of each of the agent types, we built into our project a statistics module, with parameters customizable using a properties file.
For comparing the different types of agents, we chose the same initial parameters. You can see chosen parameters in Annex A.
When it comes to the demonstration of the results, we decided to represent each line of the graphics with the average of 10 runs, having a total of 5 representative lines of the 50 runs. We will study two types of statistics, the accuracy of detection and the time for detection.

5.1 Baseline Agent

Loss percentage: 72%
Average survival time: 1562 ticks
Agent Parameters:

- Ping time: 21
- Trust threshold: 50%

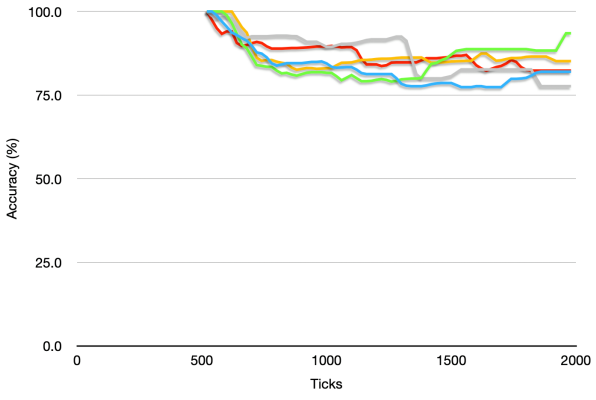

Figure 2: Average Accuracy of Detection

As expected, since the normal distribution models the ping data fairly well, the accuracy of our baseline agent appears to be fairly high and constant (around 80%).

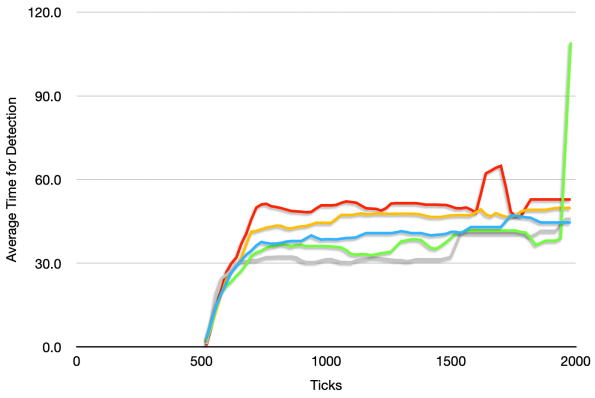

Figure 3: Time for Detection

The time for detection is also fairly good, though its seems to get worse closer to the end. A possible explanation would be the fact that points from infected servers are added to the normal distribution used by this agent, thus deviating it and causing the time for detection to become worse.

5.2 Memory Agent

Loss percentage: 64%
Average survival time: 1615.2 ticks
Agent Parameters:

- Ping time: 21
- Trust threshold: 55%
- Number of saved pings: 10
- Average difference: 2.5

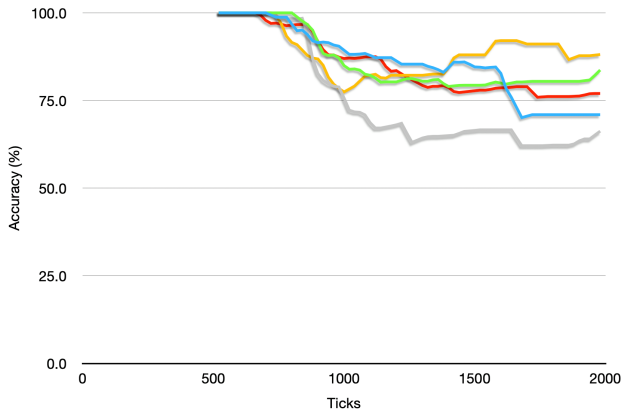

Figure 4: Average Accuracy of Detection

As we can see, the accuracy of this agent varies significantly more than the baseline agent, though it can reach higher values and the agent survives longer, on average.

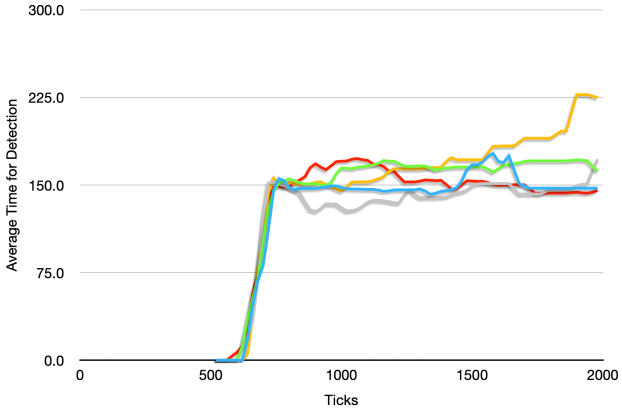

Figure 5: Time for Detection

Despite the accuracy achieving slightly higher values, the time for detection is much higher. This is due to the fact that the agent requires the average of the 10 points to change enough, and a point is added only around every 21 ticks (Ping time). Considering this information, we decided to implement a simple learning mechanism of the ping, in order to reduce this time.

5.3 Memory and Ping Learning Agent

Loss percentage: 58%
Average survival time: 1537.2 ticks
Agent Parameters:

- Trust threshold: 55%
- Number of saved pings: 10
- Average difference: 2.5

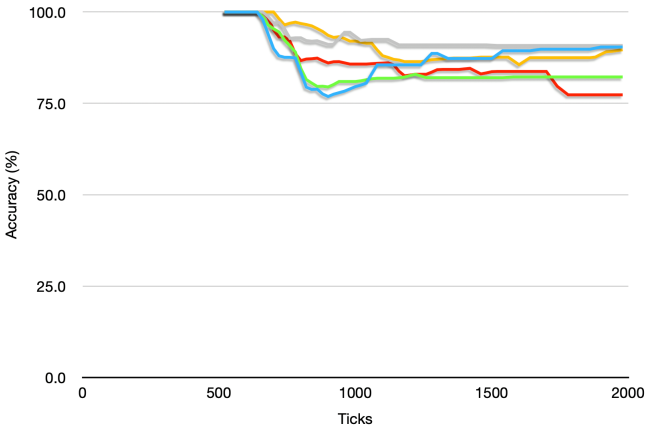

Figure 6: Average Accuracy of Detection

The accuracy of this agent, as expected, was much more constant and slightly higher than the regular memory agent. It was also slightly higher and more constant than the baseline agent, its loss percentage was also the best out of all the agents.

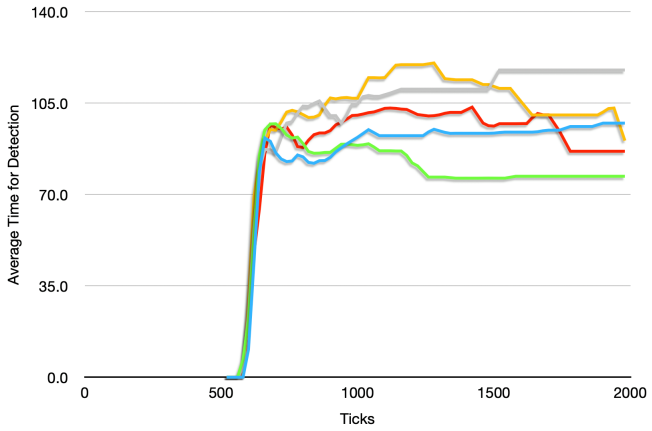

Figure 7: Time for Detection

In terms of the time for detection, we also see a clear improvement from the regular memory agent, as expected.

5.4 Perfect Agent - Cautious

Loss percentage: 64%
Average survival time: 1690.4 ticks
Agent Parameters:

- Trust threshold: 0%

This agent, since it has perfect environment knowledge and is cautious (trust threshold is 0%), achieves 100% accuracy. Even with 100% accuracy, this agent has a high loss percentage (this will be further analysed in the conclusion, since the reason is relevant for analysing the performance of all agents).

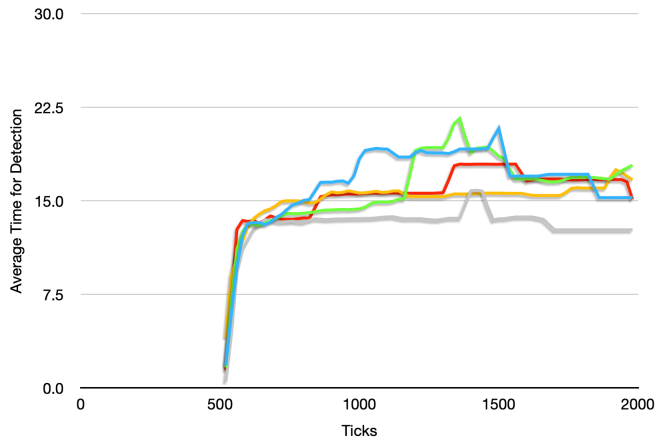

Figure 8: Time for Detection

The time for detection for this agent is extremely good, as expected from the domain knowledge that it has.

5.5 Perfect Agent - Risky

Loss percentage: 64%
Average survival time: 1611.2 ticks
Agent Parameters:

- Trust threshold: 6.4%

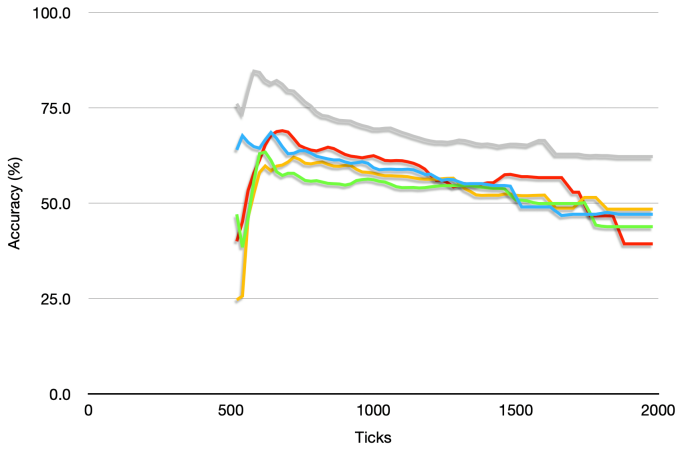

Figure 9: Average Accuracy of Detection

This agent, in case of uncertainty, since the threshold is 6.4, allows for 5 consecutive suspicious responses before requesting the removal of the pair. This results in a lower accuracy compared to the cautious agent. It is interesting to see that the loss rate for both perfect agents is higher than the Memory and Ping Learning agent.

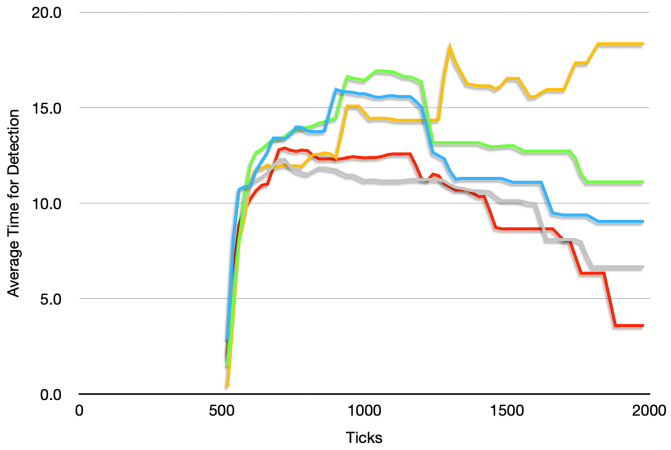

Figure 10: Time for Detection

This agent has the best time for detection out of all of the agents. More specifically, in comparison to the cautious perfect agent, the detection is better since it takes more risks.

6 CONCLUSION

The first conclusion we can draw from the results of all agents, is that the loss rates are very high.

A possible explanation is that this is a reflection of the underlying fluctuations in the network. To demonstrate this, below we have Figure 11 with the possible response times for the chosen parameters (not accounting for the distance).

Figure 11: Possible response times

The green values (2, 3, 4) are those where we can be sure that the server is healthy, since 2 is the minimum delay for a server and the infected delay is 3. Similarly, the red values (7, 8, 9) are those where we can be sure that the server is infected, since the maximum delay is 6. The main issue are the yellow values (5, 6), those where we can't be sure of the state of the server. With this amount of possibilities, the chances of any ping delay being one of the uncertain values is $\frac{2}{8} = 25\%$. With the frequency of pings being so high, the chances of this eventually happening in any given run are very high.

A possible solution for this issue, would be to make the delays more realistic, since usual delays are at least in the tens of milliseconds in magnitude. This would require significant computing power to test for a relevant amount of time, which is why we could not test this solution in this project.

Given the restraints explained above, we can say our Memory and Ping Learning agent achieves results which are fairly good given that it doesn't rely on environment knowledge (which would be unrealistic to get in a real scenario) like the Perfect agent does.

Additionally, since we built our system modularly, it would be interesting to try out other types of agents within the system and to try more realistic time frames to see how the system behaves.

A TESTED PARAMETERS

- Number of runs: 50
- Number of ticks per run: 2000
- Number of agents: 13
- Quorum size: 5
- Invulnerability time: 500
- Probability of Outside Infection: 5%
- Server - Minimum time to Answer: 2
- Server - Maximum time to Answer: 6
- Server - Infected delay: 3
- Server - Work Frequency: 3
- Server - Probability Inside Infection: 1%
- Fault Detector - Probability Inside Infection: 1%

B GRAPHICAL INTERFACE

B.1 Initial Parameters

When you run the program, you will be shown a set of parameters and that you may define before starting the simulation. Note that at any point, you can easily save a certain type of agent by going to the resources directory and adding your own Agent.properties file.

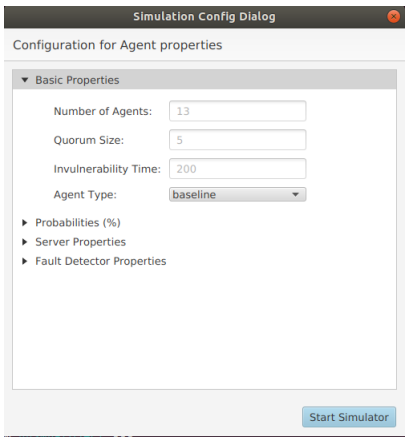

Figure 12: Graphical Interface Parameters

There are four main modules of the parameters:

- **Basic Properties**:
  - Agent Type, which has its own default parameters as you will be able to see in each entry;
  - Number of Agents, which is the number of pairs Fault Detector/server;
  - Quorum Size, which should be an odd number to ensure you don't have a draw;
  - Invulnerability Time, which represents the number of ticks before an entity can become infected.
- **Probabilities**:
  - Probability of an Inside Infection for a server;
  - Probability of an Inside Infection for a Fault Detector;
  - Probability of an Outside Infection.
- **Server Properties**:
  - Minimum time to answer when healthy;
  - Maximum time to answer when healthy;
  - Delay of when the server is infected;
  - Work frequency, which represents when a server sends a message to its neighbour, as a normal network would be.
- **Fault Detector Properties**: These properties are dependent on the type of agent chosen. For example:
  - **Baseline Agent**:
    - Trust Threshold, which represents the level of trust required in order for an agent to believe a server or fault detector is infected;
    - Ping Time, which is the interval for which a Fault Detector pings the servers;

B.2 Running the simulation

When the initial parameters have been successfully set, a new window with the representation of the pairs will appear. In this window you can start the simulation by pressing play, pause it at any time or restart it with the same parameters. In the bottom right corner, you can also choose the step intervals of the ticks. Each node can be surrounded by the colors green, representing a healthy entity; purple, representing an infected entity; or red, representing a removed entity. Finally, at any point you can double click on any fault detector to see its own statistics of accuracy of its predictions and time for detection.

REFERENCES

[1] Toueg S. Chandra, T.D. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (1996), 225–267. https://doi.org/10.1145/226643.226647
[2] Xuemin Lin. 1997. A Fully Distributed Quorum Consensus Method with High Fault-Tolerance and Low Communication Overhead. *Theor. Comput. Sci.* 185 (1997), 259–275.
[3] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com* (03 2009).
[4] Sergio Pastrana and Guillermo Suarez-Tangil. 2019. A First Look at the Crypto-Mining Malware Ecosystem: A Decade of Unrestricted Wealth. (01 2019).
[5] Mahsa Najafzadeh Daniel Porto Allen Clement Duarte Sergio Marco Carla Ferreira Johannes Gehrke Leitao Joao Carlos Antunes Preguiça Nuno Manuel Ribeiro Rodrigo Rodrigues Marc Shapiro Viktor Vafeiadis Valter Balegas, Cheng Li. 2016. Geo-Replication: Fast If Possible, Consistent If Necessary. *IEEE Data Engineering Bulletin* 39, 1 (16 3 2016), 81–92.