11

Capítulo 11

Towards a Distributed Systems Model based on Multi-Agent Systems for Reproducing Self-properties

Capítulo 11

Towards a Distributed Systems Model based on Multi-Agent Systems for Reproducing Self-properties

Arles Rodríguez y Jonatan Gómez

Abstract

In this chapter, we introduce a preliminary model of a distributed system (some algorithms and principles are modelled) which has been inspired by multi-agent systems. Agents can communicate in a local fashion and establish a cooperation process in order to compute some desired functions as a traditional distributed system, producing an output that is transparent to an end user. Each agent can sense new information, and acts following certain information in a decentralized way. As a result, the output of the implemented functions achieves the same result as a traditional distributed system by using local behaviours. Finally, we show that it is possible to define crashes in the proposed model in a local, natural and easy way. The purpose of doing this is to use this simulated model to introduce behaviours to agents that allow the system to recover from failures by itself, in turn, achieving self-* properties.

Key words: Distributed Systems, Multi-agent Systems, Model, Computation, Global output, Local input, Autonomic Computing, Crash Failures, Self-* properties, Decentralised control.

Introduction

A distributed System is a collection of autonomous components connected through a network, which coordinate their activities and share resources in order to offer services to end users in a transparent and easy way (Tanenbaum & Steen, 2006). However, distributed systems are complex to manage, expand, or maintain, since they are prone to failures, their components have partial knowledge of the environment, and can they fail, requiring adaptation to warrant completion of a determined task (Raynal, 2013).

Autonomic Computing faces complexity with the idea of a computer system that adapts to changes without human intervention (Lalanda, Mccann, & Diaconescu, 2013). Autonomic Computing defines an autonomic system as a set of autonomic elements which are responsible for managing a particular element (White, Hanson, Whalley, Chess, & Kephart, 2004). An autonomic element manages its own state and its interactions with an environment (White et al., 2004). The environment consists of signals and messages from other elements and the external world (Kephart, Chess, Jeffrey, & David, 2003). From the Self-management goal, some self-* properties emerge: adapting to the addition or deletion of components (Self-configuration) (Kephart et al., 2003), detecting and recovering from failures without disruption to the system operation (Self-healing) (Nami & Bertels, 2007), finding improvements in the efficiency of a system (Self-optimization) (Lalanda et al., 2013), and anticipating and preventing threats (Self-protection) (Lalanda et al., 2013).

Distributed systems and autonomic elements can be modelled as multi-agent Systems (Lalanda et al., 2013). Agents are designed as autonomous adaptive entities that observe their internal and external states, act based on their local perceptions, and can model multiple feedback loops(Jun et al., 2004). Additionally, cooperative agents are able to control a computer network, just based on the agent's (local) interactions, or are able to model some autonomic elements. In this way, an agent has the ability to manage resources in a network by analysing the capabilities of each element and by defining it as a managed element (Guo, Gao, Zhu, & Zhang, 2006).

In this chapter, we address the problem of modelling a distributed system as a multi-agent system and propose a failure model over the agents as a starting point to model self-* properties. The proposed model is different from traditional distributed systems simulation because it is focused on modelling local behaviours and interactions between processes in a local way instead calculating network usage or CPU as packet level simulators (Bhardwaj & Dixit, 2010) or flow level simulators (Casanova, Legrand, & Quinson, 2008; Fujiwara & Casanova, 2007; Velho & Legrand, 2009).

Through the modelling agents, it is expected to have a more natural and be closer to a living system version of a distributed system. This is be possible, because agents perform some computations in the same way as distributed systems do, but they can present failures at a local level that can be repaired by adding behaviours proper to biological systems, or social dynamics to the agents. The idea is that agents will perform the task of an autonomic manager and the network managed resource at the same time.

The remaining part of this paper is organized as follows: the next section presents some background regarding the distributed systems definition taken, then, the following section deals with the design of the agent's environment and the communications protocol. The fourth section shows the computed functions and their design. The fifth section presents some experiments regarding the computation of some functions (routing tables, max, and min) and a failures definition. Finally, some conclusions are drawn.

2. Distributed systems generalities

A distributed system can be seen as a "natural" object based on the collective work of processes connected through a communication system (network). The idea of this paper is to have a collection of agents that represent processes in which, each process can carry out a well-designed simple task, communicate the result to other processes, and obtain a complex output as a result of the collective work (Tel, 2009).

Distributed systems can be modelled as a set of processes $\Pi = \left\{p_1, p_2, \ldots, p_j\right\}$, where p_i is able to communicate with p_j and use a communication channel (Satzger, Pietzowski, & Ungerer, 2011). A basic communication protocol is based on sending and receiving messages and can be defined in each node as a buffer. This buffer contains messages that have been sent to the process, but not yet received (Satzger, 2008). Figure 1 presents the two main communication primitives (Fischer, Lynch, & Paterson, 1985; Satzger, 2008): $Send(m,q)$ sends a message $m \in M$ where M

is a fixed universe, to a process q and $Receive(q)$ cleans the buffer of process q and delivers messages to the process if the message is delivered or returning null marker '∅' otherwise. Additionally, communication channels are bidirectional and the ability to send messages between processes or nodes determines the network topology (Satzger, 2008).

There are two known types of communication in distributed systems: *synchronous* and *asynchronous* (Kshemkalyani & Singhal, 2008). On the one hand, a distributed system is synchronous if the Send primitive is blocked until the corresponding Receive primitive has been performed and completed. On the other hand, a distributed system is asynchronous if there are no timing assumptions regarding message delay, clock drift, or time taken to execute a Send primitive, and the control returns back to the invoking process after the data is copied in the buffer of the process that receives. Asynchronous Simulation of Distributed Systems is simpler and easier to replicate than synchronous models because it does not include timing assumptions (Chandra & Toueg, 1996).

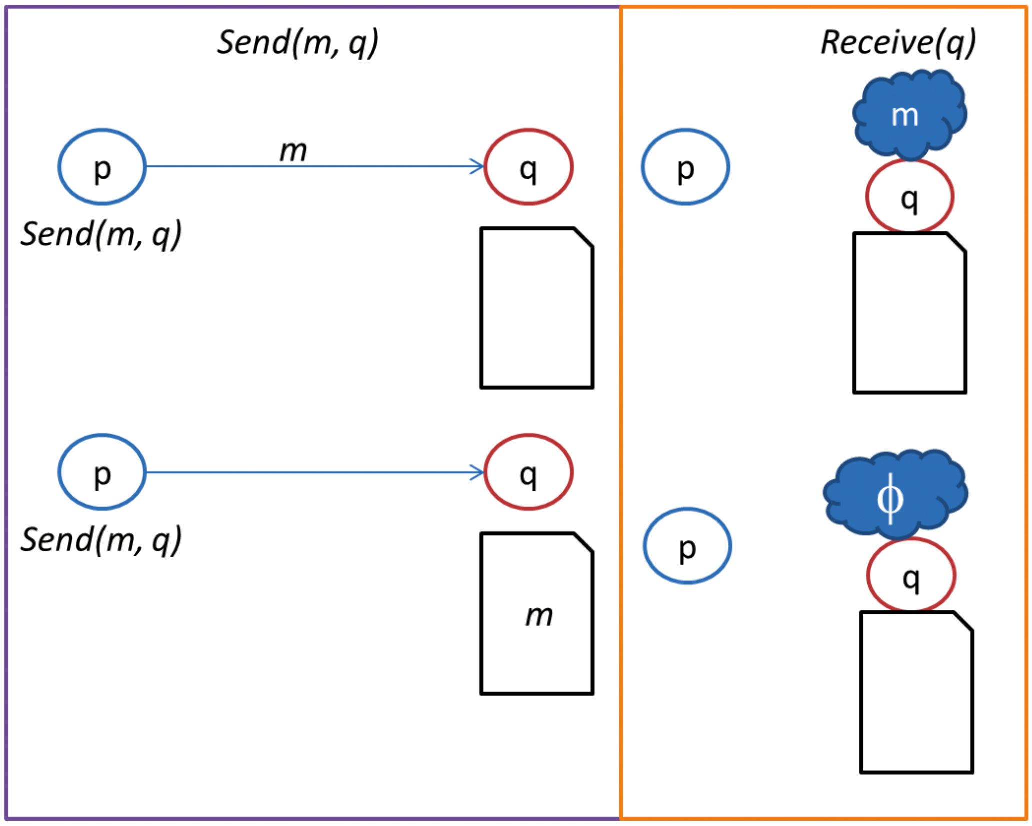

Fig. 1. Communication Primitives

3. Model based on multi-agent systems

Multi-agent systems define agents capable of independent actions with the ability of interacting with others. Agents are required to cooperate, coordinate and negotiate in order to achieve their tasks (Balaji & Srinivasan, 2010). To model a distributed system, each process p_i is defined as an agent. Each agent (process) senses information from its environment (network) and acts in this environment by using actuators (Russell & Norvig, 2004). This multi-agent model is based on the Algorithmic Framework to Compute Global Functions on a Process Graph of Raynal, 2013. Based on this proposal, the following rules are defined for the agents:

- No centralized control: Agents obey rules of behaviour and do the same task in a local way by communicating with their neighbours.
- Communication Channels are FIFO: Messages received first are processed first by each agent.
- Local communication and variables: Each agent receives and sends messages only to its neighbours. Each agent p_i has its own local inputs $input_i$ and local outputs out_i.

3.1. The Agent's Environment

A Network is a graph that stores processes, identities, and the channels that define the neighbours of each process in the form of vertices and edges. Additionally, network stores a message queue for each process to model FIFO communication channels and implements the agent's architecture, which is the way agents initialise sense and act in the environment (Fig 2). In this way, a network is defined as a dynamic and non-deterministic environment because agents have partial control over the network; they can change in the way that processes and channels can crash and therefore produce unexpected results (Russell & Norvig, 2004).

A Distributed System can be scalable in terms of size if more resources can be added to the system in an easy way and in terms of geographical distribution if different resources or network topologies are supported (Tanenbaum & Steen, 2006). A model is scalable if it is capable of implementing different network topologies and resource failures, and complex behaviours are simulated with precision in a fast way (Casanova et al., 2008).

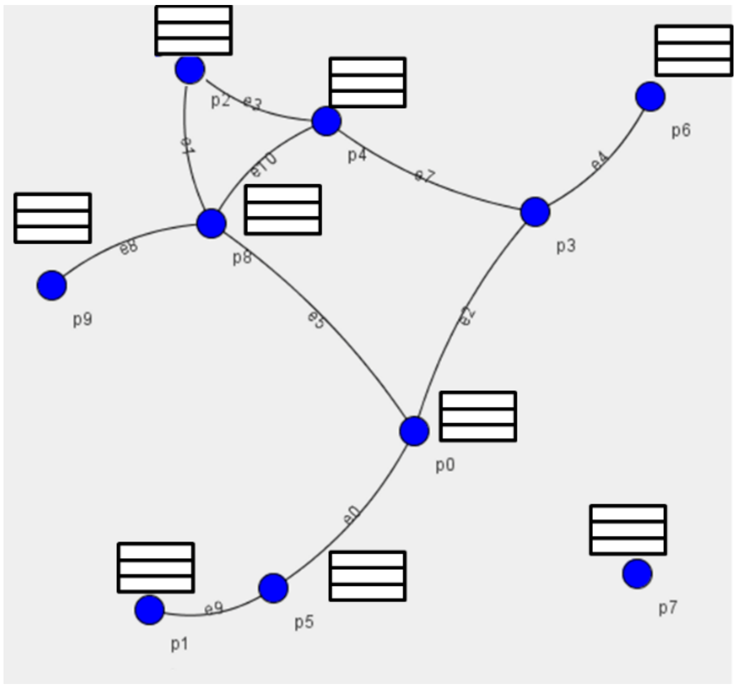

Fig 2. Network Visualisation

Scalability is also required for failure detection and healing models because these processes require, at least, the establishment of monitoring relationships by exchanging messages, and cooperation between nodes can be necessary to guarantee scalability (Satzger, 2008). Based on these ideas, a model must support different network configurations. A graph library called JUNG (White, 2005) is used to represent the network in the model. This library allows the generation of random graphs by using different algorithms from the literature.

JUNG is a Java framework used to generate different random network configurations (White, 2005). This framework implements a random generator of graphs that approximate its behaviour to power laws (Eppstein & Wang, 2002). It is chosen because some researchers observed that computer networks and internet topology exhibit a power law distribution (Eppstein & Wang, 2002; Faloutsos, Faloutsos, & Faloutsos, 1999). Additionally, by using power laws, it is possible to analyse the average-case behaviour of network protocols (Faloutsos et al., 1999).

Serialization implements the data transfer behaviours of Distributed Systems. Data structures and objects can be sent from an agent to another by serialization, and receiving processes can deserialize and create a semantic clone of the object.

3.2. Agent's Model

Agents by now are reactive. This means that each agent operates to respond to changes and to satisfy its design objectives (Russell & Norvig, 2004). An agent program defines the main thread of an agent, which determines the next action to execute according to its local knowledge. The main thread of each agent looks like algorithm 1 (Alg. 1). While an agent is alive (`status != Action.DIE`) this agent senses from the network; after that it chooses an action based on its perceptions, and finally the action has an effect on the environment.

```
round = -1;
while(status != ACTION.DIE){
      Percept p = network.sense();
      Action action = program.compute(p, round);
      Network.act(this, action);
      round++;
}
```

Alg. 1. Main thread of an agent

In this work, each agent i computes a result out_i based on its local input $input_i$. From an external point of view, there is a vector of inputs $INPUT$ such that $\forall_i \left(INPUT[i]\right) = input_i$ and a vector of outputs $(OUTPUT)$. This design is based on *An Algorithmic Framework to Compute Global Functions on a Process Graph* by Raynal, 2013. The idea is that agents cooperate and coordinate to compute $OUTPUT = F(INPUT)$. In this way, processes can compute the same output $(min, max, sorting)$, or different outputs ($out_i \neq out_j$) that give global information (e.g. routing tables of a network) for experimental purposes, or a user can request information locally from each process like in a real distributed system in a transparent way.

A key point of this proposal is the communication between agents and the time of each simulation. Each simulation step in an agent loop is a round (Alg. 1.). In each round, an agent shares its local knowledge and gets new information from neighbours. Additionally, based on the new information, it updates and computes the function *F*, and determines the new local information to broadcast to its neighbours. The process stops when there is no new information to share or if the agent fails $(status = ACTION.DIE)$.

Agent perceptions are in a collection called $perceptions = \{round,\ hasMsg,\ neighbors,\ ch_{in},\ ch_{out},\ message\}$. *round* is an agent's internal counter of iteration;

hasMsg is a boolean flag that is true if the message buffer of the agents has messages; *neighbours* return a list with the identities of the neighbours for communication; ch_{in} and ch_{out} are the list of input and output channels respectively, and *message* represents new information received from a neighbour.

An agent has the following actions: *Initialise, Send, Receive* and *Die*. Initialise sets parameters for a determined algorithm like agents identities (p_i) and custom information ($input_i$, new_i); *Send* defines propagation of new information that an agent has received previously from its neighbours; *Receive* computes a determined algorithm with the local information received and calculates what the new information is for sharing in the next round. Finally, *Die* allows each agent be able to return its local output out_i and terminate its execution thread. In contrast to the model of Raynal, 2013, the actions Send and Receive are executed given some conditions instead of run sequentially in each agent program as shown in Alg. 2. In the first round, each agent performs Initialize. An agent always performs Receive when it receives a new message; in other cases, the agent does *Send*.

```
Action compute(Percept p, int round) {
        Action act = null;
        double probFailure = 0.1;
        //process can fail with a probability of 0.1
        if (Math.random() < probFailure) {
            return new Action("Die");
        }
        if (round == -1) { //round -1 means initialize
            act = new Action("Initialize");
        } else if ((boolean) p.getAttribute("hasMsg") == true) {
            act = new Action("Receive");
        } else {
            act = new Action("Send");
        }
        return act;
}
```

Alg. 2. Actions definition

Each agent has some variables to control its internal status and to store the new information that it gets from its neighbours. inf_i stores all the information learned from neighbours, new_i stores the new information learned on each round, and out_i stores the output of the local function. Fig. 3. shows the agent's design. In the next subsections, details regarding an agent's actions and information management are drawn.

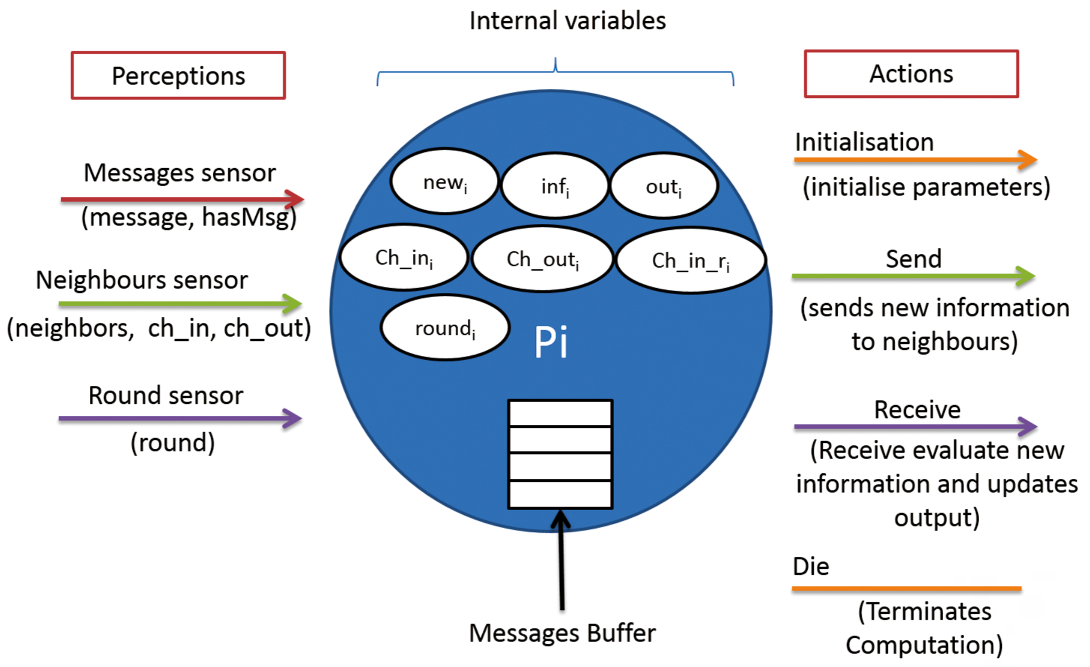

Fig. 3. Agents Design.

3.3. Initialisation

An agent i performs initialisation at the start (in the first round) by assigning values for each local input $input_i$ depending of the function F. In the first round, $input_i$ and new_i can have the same value. For organization purposes, these variables are defined as a collection structure with each element defined as the pair$\langle p_i, value_i \rangle$, where p_i is the agent id and $value_i$ is the local value. After setting variables, the round number is increased.

Each agent i has collections of input channels $Ch\_in_i = \{Ch\_in_j, Ch\_in_k, \ldots, Ch\_in_n\}$ and output channels $Ch\_out_i = \{Ch\_in_j, Ch\_in_k, \ldots, Ch\_in_n\}$ to reach its neighbours $j, k, \ldots, n$. These variables are loaded in the initialization method as a perception and stored as local variables. In this chapter, experiments use bi-directional channels, but this design allows for the definition of one-directional too. There is another collection called $Ch\_in\_r_i$ that stores the list of channels that have not sent messages to this agent in a given round. In the first round $Ch\_in\_r_i = Ch\_in_i$. In this model, $Ch\_in\_r_i \neq \varnothing$ means that there is pending information from some neighbour in some round or that it is the initialisation round.

3.4. Send and Die

An agent spreads new information to its neighbours by Send. This action allows agents to choose the information to send to a neighbour *j* by using the output channel Ch_out_j. New information to be sent to channel *j* is the difference between the new information new_i and the received information from the same channel (agent) in the last round ($Recv[j]$) (Eq. 1 from Raynal, 2013).

$$Send[j] = new_i \setminus Recv[j] \qquad \text{(Eq. 1.)}$$

Each message is encoded as a collection $M = [from|msg|chan]$, where from is the id of the agent sender, msg is the serialized content of the message, and chan is the channel from which msg was sent. Send a message is to store each msg in the FIFO queue mailbox that the network has for each neighbour. $Send[j] = \emptyset$means that there is no new information to share with this process by channel *j*, so *j* is removed from Ch_out_i after an agent sends $\emptyset$. Once the new_i is sent by all the channels $new_i = \emptyset$. Finally, the round number is increased by one.

After the round number is increased and messages are sent to all the neighbours, $Ch\_in_r = Ch\_in_i$. This means that an agent has shared information and now is the moment for receiving information from all its neighbours. In the opposite way, messages are sent, if and only if, $Ch\_in_r = \emptyset$. This means that only new information is shared when all the neighbours have reported information to the agent or the agent has been initialized.

Additionally, the logic expression $Ch\_in_i \neq \emptyset \wedge Ch\_out_i \neq \emptyset$ which represents that there is no new information to be sent or received is evaluated. If this condition is true, the agent returns the out_i to a control board that stores the results of all the agents and finally ends the agent process. Die is an action that sets the agent status in `Action.DIE` and it stops the main thread of the agent.

Failures are defined in this work with the action `Action.DIE` as shown in Alg. 2. A failure probability $probFailure \in [0,1]$ is defined. For example, a $probFailure = 0.1$ means that an agent would stop in one of ten actions.

3.5. Receive

This action allows agents to receive new information from others and to compute local outputs. First, each agent decodes the received message $msg = [from|msg|chan]$ and evaluates the difference between the information received by the channel $chan$, defined as *Recv[chan]* with the information sent using *chan*. As Raynal, 2013 remarks, there are five possible scenarios:

- if $Send[chan] = Recv[chan]$, agents that are connected by channel *chan* has the same information at this point. So *chan* must be removed from Ch_in_i and Ch_out_i.
- if $Recv[chan] = \emptyset$, *chan* must be removed from Ch_in_i.
- if $Send[chan] \subseteq Recv[chan]$ the agent that sent the message has more information, so chan must be removed from Ch_out_i.
- If $Recv[chan] \subseteq Send[chan]$ the agent that sent the message has less information, so it must be removed from Ch_in_i.
- In other examples, case agents learn from each other them because they have complementary information.

After evaluating the channels, each agent proceeds to do its task. This task is the computation of the function *F* which takes the message *msg* received, processes *msg*, produces an output and stores its result in out_i $(out_i = F(msg))$. Once F is computed, new_i and inf_i are determined by Eq 2 and Eq 3 like in Raynal, 2013. These equations mean that new information is the union between the new information received by the neighbours in this round new_i and the difference between the received information and all the information that an agent has learned inf_i. Finally, the channel from *msg* which was received is removed from Ch_in_r.

$$new_i = new_i \cup (Recv[j] \setminus inf_i) \qquad \text{(Eq. 2)}$$

$$inf_i = new_i \cup inf_i \qquad \text{(Eq. 3)}$$

4. Experiments and results

The purpose of the experiments is to execute a function F in a network with failures and compare the output with the global expected output. Network is generated using the Eppstein Algorithm of JUNG (Eppstein & Wang, 2002; S. White, 2005). Parameters to generate a Network are the number of nodes n, the number of edges e, the function to compute F, and l the number of iterations to approximate a power law topology.

An experiment is a combination of parameters performed 30 times as follows: $F = min, n = 50, e = 150, l = 1000, probFailure = 0, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}$ and 10^{-1}. The idea of these experiments is to see how crashes can affect correctness and communication. Network used in experiments as shown in Fig 4b.

4.1. Implemented functions

In this work functions *max*, *min*, *sorting values* and *routing tables* are implemented. Definition of functions are local and executed as *F* in the *Receive* action of each agent. Agents take inf_i and new_i and assign values to out_i. Implementing each algorithm is easy because they only imply to change the value (or type) of new_i, inf_i and out_i and establish a predicate.

Table 1 presents a description of the different functions computed. Declarations of variables of type collection are written as $var \in \langle type \rangle []$. Identity of process i (p_i) is a *String* $(p_i \in String)$. \ and ∪ are difference and union of collections respectively. In the case of max and min, the algorithm is performed over the data of *Integer* type, so there is classic calculation of max and min having each process in a network a value. The same case happens with sorting where an *Integer* value of each process is extracted and added to an *Integer* collection *Integer*[] and this collection is sorted.

Table 1. Functions F Implemented.

Function	Parameter	Definition
Max	Input type	$inf_i \in < p_i, Integer > [], new_i \in < p_i, Integer > []$
	Predicate	$aux < p_i, Integer > [] = \mathrm{Re}\,cu[ch] \setminus (new_i \cup inf_i)$ $out_i = max(aux, inf_i)$
	Output type	$out_i \in < p_i, Interger >$
Min	Input type	$inf_i \in < p_i, Integer > [], new_i \in < p_i, Integer > []$
	Predicate	$aux < p_i, Integer > [] = \mathrm{Re}\,cu[ch] \setminus (new_i \cup inf_i)$ $out_i = min(aux, inf_i)$
	Output type	$out_i \in < p_i, Integer >$
Sorting	Input type	$inf_i \in < p_i, Integer > [], new_i \in < p_i, Integer > []$
	Predicate	$aux < p_i, Integer > [] = Recu[ch] \setminus (new_i \cup inf_i)$ $out_i = sort(aux \cup inf_i)$
	Output type	$out_i \in < p_i, Integer[] >$
Routing Tables	Input type	$inf_i \in < p_i, p_i > [], new_i \in < p_i, p_i > []$
	Predicate	$aux < p_i, p_i > [] = Recu[ch] \setminus (new_i \cup inf_i)$ $out_i \cdot put(ch, (out_i \cdot get(ch) \cup aux))$
	Output Type	$out_i \in < ch, p_i[] > []$

In the case of routing tables, inputs are agents' identities and the outputs are channels that an agent p_i can use for communicating with its neighbours. *get* and *put* are methods that return or insert a collection of data associated to a channel e_j (in this case, the list of agents reachable from channel e_j). For example, the input for the process p_6 - it is the red process in the network of Fig. 4a - is $inf_6 = \langle p_6, p_6 \rangle, new_6 = \langle p_6, p_6 \rangle$, the value computed of this output is: $out_6 = \{e_{29} = \{p_{10}\}, e_{14} = \{p_9, p_{11}\}, e_{11} = \{p_4, p_1, p_0\}, e_3 = \{p_8, p_7, p_{14}, p_5, p_{13}\}, e_0 = \{p_{12}, p_3, p_2\}\}$.This output can be interpreted as: p_6 reaches p_{10} by using channel e_{29}, by e_{14} process reaches p_9 and p_{11}, etc.

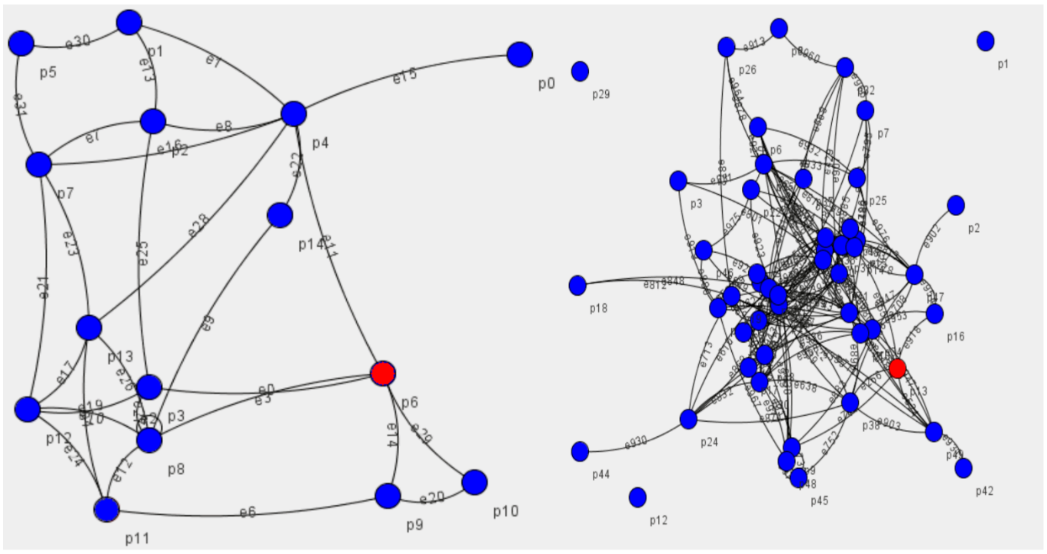
a) Routing Execution Example b) Network used in experiments

Fig. 4. Processes Networks

4.2. Results

Table 2 presents the result of the experiments performed. Each row represents the average and the standard deviation of agents which have a correct output (Agents Right), agents with a wrong output (Agents Wrong) and agents that do not report an answer (Agents without Response). As shown in Fig 4b, the network used for experiments is not fully connected, so it is expected to have 47 processes communicating among themselves in order to achieve the global task. Table 2 shows how the model computes F in an accurate way if $probFailure = 0$. A value of 3 in the column *Agents without Response* represents the agents p_1, p_{29} and p_{12} which have no input or output channels and do not send a response to the control board (in the Network of Fig. 4b).

For a greater value in the *probFailure* column, more agents with wrong answers and agents without response are observed. In the experiments performed, it is observed that the main effects of a crash are wrong answers more than problems in communication. Agents without results represent agents that crash without receiving a message because each time an agent receives a message, it reports its output to the control board. Additionally, it is observed (by now empirically) how the crash of a process increases execution times because this implies that other agents remain waiting for a response for a long time until they crash (given a *probFailure*). Lower values in the *probFailure* column imply less processes with failures and more process waiting for answers from its crashed neighbors. Another interesting point is that there are few experiments where all the agents give wrong answers.

Table 2. Summary of experiments for crash ($F = min, n = 50, e = 150, l = 1000, probFailure = 0, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}$ and 10^{-1})

probFailure	Agents Right	Agents Wrong	Agents without Response
0	47 ± 0	0 ± 0	3 ± 0
10^{-6}	46.55 ± 1.5	0.45 ± 1.5	3 ± 0
10^{-5}	42.27 ± 4.09	4.10 ± 4.20	3.63 ± 0.72
10^{-4}	24.29 ± 9.53	18.39 ± 9.67	7.32 ± 1.78
10^{-3}	16.33 ± 9.99	22.63 ± 9.79	11.03 ± 3.01
10^{-1}	2.13 ± 3.21	25.03 ± 3.93	22.83 ± 4.22

As Raynal, 2013 proves, this communication mechanism allows information to be shared by all the network from process to process in the next rounds. In this case, computation terminates in maximum $D+1$ rounds, with being D the diameter of the communication graph. The Result is $D+1$ because it includes the sending $Send(ch) = \varnothing$ when there is no new information to share, and in this case, the channel is deleted from Ch_in and Ch_out. Each process is sending and receiving a message by a number of channels e in each round, so the maximum number of messages is $2e(D+1)$. Therefore, an achievement of this work is that the model proposed computes with the same number of rounds as a Distributed System for the communication protocol implemented without failures.

Conclusions

A model that represents behaviours of Distributed Systems in a multi-agent environment for testing self-* properties has been proposed. This approach defines an asynchronous decentralized protocol that models the perceptions and actions of the agents to interact with others and compute functions as a distributed system. In the original implementation of the communication protocol of Raynal, 2013 all the actions are defined together as a program for the process p_i. Each process must explicitly wait to receive a message from the other channels before starting a new round. By abstracting each process as an agent, it is possible to add more actions than an agent requires or even modify the agent program, because synchronization is achieved between the send and the receive events by storing the nodes that have not reported information in a given round.

One of the promising things of the communication protocol used is the possibility of modelling different functions in an easy way. As Table 1 presents, it is possible to define function F as a simple predicate and to set different input and output types without changing the protocol. In the processing of this actions it is possible to simulate what happen if an agent's response is incorrect (Satzger, 2008).

FIFO communication channels are easy to implement in an environment like the one proposed in this paper. The message queue model allows agents to share information in an effective way and serialization allows agents to send and receive different types of data over the network without modifying the environment. By having one buffer FIFO in the environment (for each agent) and another message buffer inside each agent, it is possible to emulate known types of failures of Distributed Systems like omission failures where a node can fail to send or receive messages or requests which can imply a loss of communication channels (Satzger, 2008; Tanenbaum & Steen, 2006).

Our model is designed in a way that custom agent programs can be defined in order to add more behaviours to the agents and additional properties can be added to the environment in an easy way, just like in the crash failure presented in this paper. Even the communication protocol can be redefined in order to adapt it to other networking models, and in the same way new perceptions and actions can be added. As future work, we hope to define different kinds of tasks in the same network to achieve self-organization and to add parameters like the quality of the network connection to the channels.

In the experiments performed, crash failures were introduced in the behaviour of the agents. The introduction of this type of failure allows for the observing of how failures are propagating into the system and how a crash influences a wrong answer or failures in the communication. As future work, we will define more experiments regarding the crashing of a process and other types of failures (response, omission). Once these experiments are defined, the idea is to evaluate what kind of local behaviours can be added to agents in order to allow for the adaptation of the system by itself and to give agents self-* properties. Additional metrics like time in rounds versus *probFailure* or message consumption are part of the future work.

By modelling decentralized control approaches based on local interactions, it is possible to see how different random networks compute good local results and how the agents without channels die because they have no other process for communication and cooperation. Source code and executable files are available at https://github.com/arleserp/DSSimulator.

References

Balaji, P. G., & Srinivasan, D. (2010). An introduction to multi-agent systems. *Studies in Computational Intelligence*, *310*, 1–27. http://doi.org/10.1007/978-3-642-14435-6_1

Bhardwaj, R., & Dixit, V. S. (2010). An overview on tools for peer to peer network simulation, *1*(19), 70–76.

Casanova, H., Legrand, A., & Quinson, M. (2008). SimGrid: A generic framework for large-scale distributed experiments. *Tenth International Conference on Computer Modeling and Simulation (Uksim 2008)*, 126–131. http://doi.org/10.1109/UKSIM.2008.28

Chandra, T., & Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, *43*(2). Retrieved from http://dl.acm.org/citation.cfm?id=226643.226647

Eppstein, D., & Wang, J. (2002). A steady state model for graph power laws. *2nd Int. Worksh. Web Dynamics*, 8. Retrieved from http://arxiv.org/abs/cs/0204001

Faloutsos, M., Faloutsos, P., & Faloutsos, C. (1999). On power-law relationships of the Internet topology. *ACM SIGCOMM Computer Communication Review*. http://doi.org/10.1145/316194.316229

Fischer, M. J., Lynch, N. a., & Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, *32*(2), 374–382. http://doi.org/10.1145/3149.214121

Fujiwara, K., & Casanova, H. (2007). Speed and accuracy of network simulation in the simgrid framework. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*. Retrieved from http://dl.acm.org/citation.cfm?id=1345279

Guo, H., Gao, J., Zhu, P., & Zhang, F. (2006). A self-organized model of agent-enabling autonomic computing for grid environment. *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, *1*, 2623–2627. http://doi.org/10.1109/WCICA.2006.1712837

Jun, H., Ji, G., Zhongchao, H., Beishui, L., Changyun, L., & Jiujun, C. (2004). A new rational model of agent for autonomic computing. *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, *6*, 5531–5536. http://doi.org/10.1109/ICSMC.2004.1401074

Kephart, J. O., Chess, D. M., Jeffrey, O., & David, M. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41–50. http://doi.org/10.1109/MC.2003.1160055

Kshemkalyani, A., & Singhal, M. (2008). *Distributed computing: principles, algorithms, and systems*. Cambridge University Press. http://doi.org/CBO9780511805318

Lalanda, P., Mccann, J. A., & Diaconescu, A. (2013). *Autonomic Computing: Principles, Design and Implementation*. Springer.

Nami, M. R., & Bertels, K. (2007). A survey of autonomic computing systems. *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, 26. http://doi.org/10.1109/CONIELECOMP.2007.48

Raynal, M. (2013). *Distributed Algorithms for Message-Passing Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-642-38123-2

Russell, S., & Norvig, P. (2004). *Inteligencia Artificial. Un enfoque moderno. 2da Edición*. http://doi.org/M-26913-2004

Satzger, B. (2008). *Self-healing Distributed Systems*. Augsburg University. Retrieved from http://www.infosys.tuwien.ac.at/staff/bsatzger/publications/pdf/satzger_diss.pdf

Satzger, B., Pietzowski, A., & Ungerer, T. (2011). Autonomous and scalable failure detection in distributed systems. *International Journal of Autonomous and Adaptive Communications Systems*, *4*(1), 61. http://doi.org/10.1504/IJAACS.2011.037749

Tanenbaum, A., & Steen, M. Van. (2006). *Distributed systems: Principles and paradigms*. Prentice-Hall. Retrieved from http://dl.acm.org/citation.cfm?id=1202502

Tel, G. (2009). *Introduction to Distributed Systems*. Cambridge Press. Retrieved from http://dl.acm.org/citation.cfm?id=517021

Velho, P., & Legrand, A. (2009). Accuracy study and improvement of network simulation in the SimGrid framework. *Proceedings of the Second International ICST Conference on Simulation Tools and Techniques*. http://doi.org/10.4108/ICST.SIMUTOOLS2009.5592

White, S. (2005). Analysis and visualization of network data using JUNG. *Journal Of Statistical Software*, *VV*(Ii), 1–35. Retrieved from http://www.citeulike.org/group/206/article/312257

White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., & Kephart, J. O. (2004). An architectural approach to autonomic computing. *Autonomic Computing, 2004. Proceedings. International Conference on*, 2–9. http://doi.org/10.1109/ICAC.2004.1301340