UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

Implementing failure detection algorithms on the ADD-model with small messages

T E S I S

QUE PARA OPTAR EL GRADO DE:
DOCTORA EN CIENCIA E INGENIERÍA DE DE LA COMPUTACIÓN

P R E S E N T A:

KARLA ROCÍO VARGAS GODOY

Director de tesis: Dr. Sergio Rajsbaum Gorodezky
Instituto de Matemáticas, UNAM

Cd. de México, Febrero 2021

Agradecimientos

Ha sido un camino largo y difícil, me gustaría agradecer a muchas personas por diferentes cosas mientras realicé esta tesis.

Agradezco mucho a Armando Castañeda por ser parte de todo desde hace mucho tiempo, por todas las charlas de trabajo, por todo el amor y apoyo brindado, porque nadie me ha echado tantas porras como tú, por quitarme algunas cargas cuando ya era demasiado peso, por adoptar a Honey y quererla casi tanto como yo. Te amo mi amor.

A mis padres, Maricela y José Antonio, gracias por brindarme la oportunidad de estudiar, sin su apoyo no hubiera llegado hasta aquí, gracias por estar presentes en mi vida. A mi hermano Ricardo, por enseñarme tantas cosas de mi misma.

A Mey Lam y Sergio Montserrat por ser como mis segundos padres, los desayunitos, las cenas con pan y las pláticas largas.

A mis querida sis Diana Montes y a Víctor Maya, ¡es la tercera tesis en la que aparecen! Así de importantes son en mi vida. A mi querida Anita Vásquez, gracias por tu sincera amistad y por ser la mejor roomie ever.

A mis amigos Carlos Báez, Mariana Gleason, Armando Ballinas, Manuel Alcántara, Jimena y Tamara Montserrat por los ratos de diversión, vacaciones y tragos. A mis primis Ney, Ash y Danny por adoptarme como su cuarta hermana. Mis queridos tíos Marina y Roberto, mis otros padres postizos.

A la Dra. Elisa Viso, siempre le estaré agradecida por su apoyo al inicio de todo.

Un muy especial agradecimiento a mi asesor el Dr. Sergio Rajsbaum. Gracias por motivarme y por invitarme a trabajar, a pesar de todos los altibajos, no pude haber tenido mejor asesor. Gracias por enseñarme tanto y por tener tanta paciencia a mi negatividad. Me alegra que esta relación alumno-tutor haya funcionado, no puedo estar más contenta.

A mi comité tutor conformado por el Dr. David Flores y el Dr. Francisco Hernández, gracias por la atención brindada a mi trabajo durante estos años. A mi sinodal el Dr. Ricardo Marcelín, gracias por sus importantes anotaciones.

I would like to thank professor Michel Raynal for letting me work with him and for his support when I was looking for a postdoc.

A CONACYT por pagar mis estudios de doctorado, al proyecto PAPIIT IN106520 y a mi adorada UNAM. Al personal de la coordinación del posgrado, sobre todo a Lulú por su gran apoyo durante la pandemia.

Contents

Chapter 1

Introduction

A *distributed system* is a set of computers or processes, that cooperates and coordinates for solving a common problem [46]. Typically, they exchange information and coordinate their actions one way or another. For this thesis, we are going to concentrate on systems whose computers or processes (processes from now on) are physically apart and communicate with each other by sending and receiving *messages* through *channels*. Channels can be reliable or not, or further assumptions about it can be made, for example, if channels are *first in first out* (FIFO) or they can lose an arbitrary number of messages, etc.

Processes can be connected to each other or not. A distributed system can be represented by a connected undirected graph with the set of processes being the vertices and the set of channels being the edges. For this work, we are interested in arbitrary graphs, for being closer to real-life scenarios, and spanning trees for being a weak enough topology that keeps connected the network including all the processes.

A distributed system is said to be *asynchronous* if there is no fixed upper bound on how long it takes for a message to be delivered or how much time takes for a process to execute one step [8].

When designing algorithms for distributed systems, it is necessary to consider all the aspects of the systems including the number of processes participating, the way processes are going to communicate with each other, the reliability of the communication channels, synchrony in the system, etc. In this work, those aspects are a very important part as it will be noted in later chapters.

1.1 Consensus

Many applications in real life require processes to synchronize and agree on a common decision like an output or the next step to take, for instance, bank systems, ticket sale systems, etc. But synchronize and agree on a common decision is not only important in real-life applications, but it is also important in theoretical computer science. One of the most important problems in the distributed and concurrent computing area is the *consensus*, in which every process decides a value previously proposed by some process

and the decided value is the same for every process.

Its importance comes from the possibility of using some concurrent objects for solving consensus and the *universality of the consensus*, that explained in a very superficial way, is that given sufficiently enough concurrent objects of the kind that can be used for solving consensus, one can construct an implementation of any concurrent object with a sequential specification [29].

Unfortunately, it has been shown that this problem cannot be solved in asynchronous systems with even just one failure [24]. The main argument of this result is that a process can't distinguish if another process has crashed or it is only very slow.

Given the importance of consensus and its impossibility result, a lot of different approaches have been proposed to circumvent this result: randomization [10, 9], partially synchronous models [22], set agreement [14], unreliable failure detectors [12], among others. This thesis concentrated on the last one.

1.2 Failure detectors

Chandra and Toueg proposed the *unreliable failure detectors* [12] model as an external module that can provide to every process information about the status of the processes. Failure detectors were proposed to augment the asynchronous model of computation with the ability to distinguish, probably with mistakes, if a process has crashed or it is only very slow. Surprisingly enough, even with a considerable number of mistakes, they can be used for solving the consensus problem.

For solving the consensus using a failure detector, every process has available a failure detection module that can be queried about the status of the processes. Then, over asynchronous rounds every process proposes a value that is communicated to each process and a process waits for another process value if it is not suspected to have failed by querying the failure detection module.

Assuming that we have a failure detector that do not make any mistakes, namely, a *perfect* failure detector, it can be used for solving consensus in any system [18]. Unfortunately, it is impossible to design failure detectors that do not make any mistakes in realistic situations, since it takes time for a process to answer a query and it is impossible to distinguish a crashed process from one whose answeris still to arrive [39]. However, it is possible to implement failure detectors that can make mistakes and eventually remain correct.

Failure detectors can make an arbitrary number of mistakes by not identifying crashed processes or by wrongfully suspecting correct processes. The type of mistakes that can be done is defined by two properties: *Completeness*, defining how and when crashed processes are suspected and *Accuracy*, defining how and when correct processes are not suspected. In [12] two types of completeness and four types of accuracy are defined, giving place to a hierarchy of eight failure detectors. It has been shown that even with the weakest failure in the hierarchy proposed by Chandra and Toueg, the Consensus problem can be solved [11] as it is explained in more detail in Chapter 2.

In this work, we are concentrated in designing algorithms with small messages for

implementing failure detectors. In particular, we are interested in the *eventually perfect* ($\Diamond P$) class which can provide incorrect information for a finite time, but eventually, it stabilizes and provides perfect information permanently which is interesting enough because:

1. $\Diamond P$ can be implemented in real systems [39]
2. There is a lot of work in complete networks (as it is seen in Chapter 2), but less in arbitrary networks
3. Despite the initial mistakes, it can be used for solving the consensus [11]

Then, we move to the weaker problem *The Eventual Leader*(Ω) failure detector, which is the weakest failure detector that can be used for solving consensus as it is explained in Chapter 2. A failure detector of class Ω provides the name of a process that is still working and eventually the name is the same for all participating processes.

1.3 Related work

Initial research on failure detector implementations concentrated in the case where there is a direct link between each pair of processes. More recently there has been interest in failure detectors for arbitrarily connected networks, given that real networks are not fully connected.

Notice that using a routing algorithm to simulate a fully connected network is problematic because the routing algorithm may need information about crashes, precisely the information a failure detector provides. Additionally, a routing layer increases the uncertainty about timing and may hinder the performance of the failure detector.

For arbitrarily connected networks, the definitions of completeness and accuracy need to be extended [2]. Informally, strong completeness requires that each process eventually suspects all processes that are not in its partition, while eventual strong accuracy requires the failure detector of every processor to eventually stop suspecting all processes that are in its partition. Further generalizations to dynamic networks can be found in [26].

Hutle [32] proposed a $\Diamond P$ implementation, for arbitrarily connected networks, in a model where processes do not need to know a bound on the communication delay between arbitrary processes but only a bound on the jitter on communications between neighbors.

A different, very weak model of *ADD channels* was proposed by Sastry and Pike [48], as a realistic partially synchronous model of ill-behaved channels that can lose and reorder messages. Each channel guarantees that some subset of the messages sent on it will be delivered in a timely manner and such messages are not too sparsely distributed in time. More precisely, for each channel there exist constants K, D, not known to the processes (and not necessarily the same for all channels), such that for every K consecutive messages sent in one direction, at least one is delivered within time D.

Sastry and Pike described a $\Diamond P$ implementation for a fully connected network, which was later extended to an arbitrarily connected network of ADD channels by Kumar and Welch [34].

Implementations are usually based on heartbeat style failure detectors in which every fixed amount of time, a message is sent to every neighbor [2]. The implementation by Hutle [32] mentioned above works even if all processes are not known in advance, and hence definitions of completeness and accuracy properties need to be extended appropriately. In this implementation, every process p_i has, for every other process p_j it knows, a heartbeat table including heartbeat counters and distance counters, and hence the size of a heartbeat message is $O(\log n + \log t)$ bits, where n is the number of nodes, and t is the round number, which is unbounded. Therefore, the size of the heartbeat messages is unbounded. The implementation of Kumar and Welch [34] also uses heartbeats, but uses more detailed path information, to achieve messages of bounded size. However, the size is exponential in n, namely $O((n+1)!)$ bits.

1.4 Contribution

Our motivation was to find an implementation of $\Diamond P$ using messages of size polynomial in n, the number of processes participating (and independent of t, the number of rounds), in an arbitrary connected network. We select for our implementation the ADD model, which is explained in more detail in Chapter 3, as an interesting realistic model to test our ideas, but we extend our work to other models, in particular to networks with unknown membership, in which processes do not require knowledge on the total number of processes. Then, we wanted to reduce the message size even further, by moving from $\Diamond P$ to the weaker Ω problem, assuming a network with weaker constraints.

As it is explained in Chapters 4 and 5, in this work we concentrate on the design of failure detectors on partially synchronous systems where the communication is very weak and not all processes are communicated with each other. We focus on providing an implementation that sends messages of bounded size improving on the previous bounds on the current implementations of $\Diamond P$ and Ω by using a novel technique well known before but not used for failure detection: *Time-To-Live* (TTL) values, which are commonly used for limiting the lifetime of packets in a network or for the number of hops a packet can take [35, 51].

Designing algorithms for failure detection is not trivial. Usually, failure detectors use *timeouts* for determining if a process crashes. If very large timeouts are chosen, the system has less false suspicions, but its overall performance is slowed down. Smaller timeouts increase performance at the cost of mistakes. It is difficult to perform experiments to estimate good timeouts for many reasons, including the well-known phenomenon of very high variability of delays in practice. As it its explained in Chapter 5, this problem can be eased by the election of some parameters.

This thesis presents implementations of the failure detectors $\Diamond P$ and Ω, both implementations using the same networking technique of TTL values for achieving small messages in an arbitrarily connected network of ADD channels and implementations in

networks with unknown membership, where processes know nothing about the network.

It is worth to mention that an easy way to implement Ω consists of implementing $\Diamond P$ first and then to output the smallest identity of a non-faulty process. But in particular, the implementation of $\Diamond P$ requires a system with stronger conditions [12], so in this work we propose another approach for implementing Ω in a weaker system than the one required for $\Diamond P$.

1.5 Organization of the thesis

Chapter 2 gives us a general view of the Failure Detector area, by introducing the original Chandra and Toueg hierarchy, showing algorithms for solving the consensus using a failure detector and presenting more related work to $\Diamond P$ and Ω, that were the main motivation for this work.

Chapter 3 introduces the formal model and the ADD channels. Chapter 4 presents two algorithms that use small messages for implementing $\Diamond P$ in a partitionable network and networks with unknown membership. Chapter 5 presents the *Span-Tree* model, a condition for the network to be connected despite failures, and two communication efficient algorithms for implementing Ω in networks assuming the *Span-Tree* model and networks with unknown membership. Conclusions are presented in Chapter 6.

Chapter 2

Failure Detectors

In the consensus problem every process p_i proposes a value v_i and after some finite time, every process must decide a value v_j such that v_j was proposed by some process and v_j is the same for all processes participating. In the shared memory model, one can use instances of some sequential specified objects in order to solve consensus. This is used to evaluate the *coordination power* of those objects, better known as *synchronization primitives* [30]. Furthermore, one can use objects that solve consensus for implementing any concurrent object [30].

One of the most important results in the distributed computing area is the impossibility of the consensus in asynchronous systems with even just one failure [24]. This result even has a name: The *FLP* impossibility result after Fisher, Lynch and Patterson. The main argument of the *FLP* result, is that a process cannot distinguish if another process has crashed or it is only very slow. So Chandra and Toueg defined a mechanism to circumvent this problem: an *unreliable failure detector* [12] which is an external module that can provide to every process information about the status of the processes.

Failure detectors are characterized by two properties. *Completeness* means that every correct process eventually suspects every crashed process and *accuracy* that restricts the mistakes that the failure detector can make.

Instead of focusing on a particular implementation of the failure detector (involving the network topology, timing assumptions, message delays, etc.) it is only needed to focus on the abstract properties it satisfies for a long enough period that allows the algorithm to solve the problem. When designing an algorithm that uses a failure detector as a module, it is assumed that the failure detector satisfies properties related to the detection of failures and the algorithm is proved to be correct only assuming those properties.

It is natural to ask which conditions must satisfy an underlying system for implementing a failure detector. As it is explained later, some failure detectors cannot be implemented on certain systems, for example, a purely asynchronous system, because that would be a contradiction of the FLP result.

In this chapter, we concentrate on three main lines of research as it is stated in

the mindmap in Figure 2.1. The first line (purple nodes in mindmap) deals with the original hierarchy proposed by Chandra and Toueg as with the reductions between failure detectors, because that is the base for building the hierarchy. The second line deals with how to use the failure detector abstraction for solving certain problems (yellow nodes in mindmap), in particular for solving the consensus. For some problems, some failure detectors not in the hierarchy of Chandra and Toueg have been proposed, and it has been shown that some of them are the weakest for solving certain problems. The third line concerns the design of algorithms for failure detection (teal nodes in mindmap). A lot of failure detectors of the original hierarchy have been designed for weaker systems. For some of them, it is needed to extend the completeness and accuracy properties due to the assumptions of the underlying system. Once an algorithm is proposed, we would like to know if it is efficient.

Following the mindmap showed in Figure 2.1, this chapter is divided as follows. Section 2.1 introduces formally the Chandra and Toueg hierarchy defined in [12]. Section 2.2 shows how to use different failure detectors for solving the consensus and shows other classes of failure detectors not in the original hierarchy. Some network models that are considered for implementing failure detectors and some implementations of failure detectors can be found in Section 2.3.

2.1 The Chandra and Toueg hierarchy

Chandra and Toueg defined unreliable failure detectors in [12] as a module that can be queried by the processes about the operational status of every participating process in the network. In the same paper two types of completeness and four types of accuracy are defined. All of them differ in how much information the failure detector must provide, how much mistakes it can make and when this information must be given correctly.

2.1.1 Model

The model considered in [12] consists of asynchronous systems in which there is no bound in the message delay or the clock drift. The system consists of a set of n processes $\Pi = \{p_1, p_2, ..., p_n\}$. Every pair of processes is connected by a reliable communication channel. A process is *correct* in an execution if it does not crash; otherwise, it is faulty. The number of processes that can fail in the system is denoted with $f < n$ and the number of processes that actually crashes during an execution is $t \leq f$.

Every process has access to a failure detector that it can query. When a failure detector module is queried, it returns an array of processes that are suspected to be failed. Formally, p_i *suspects* p_j at time t if only if $suspect_i[j] = true$ at time t.

2.1.2 Failure detector classes

The two completeness and four accuracy properties stated by Chandra and Toueg in [12] are as follows.

Definition 1 (Completeness)**.** *The two types of completeness are defined as follows.*

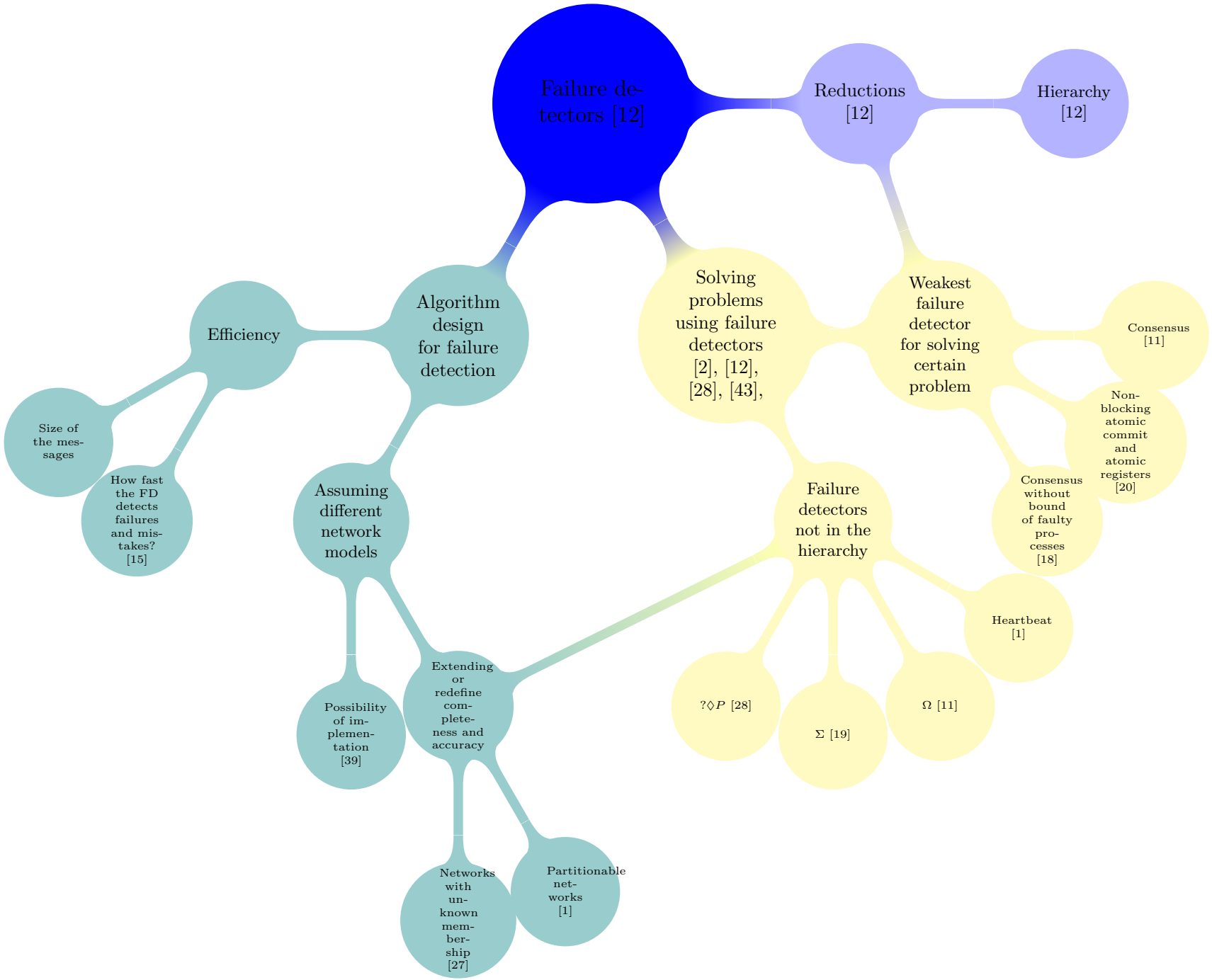


Figure 2.1: Failure detector research lines

1. ***Strong Completeness****: Eventually every process that crashes is permanently suspected by* every *correct process*

2. ***Weak Completeness****: Eventually every process that crashes is permanently suspected by* some *correct process*

It is easy to design a failure detector that satisfies one of the completeness properties. Let us consider a failure detector that marks as suspected every process in the network. That failure detector satisfies completeness (strong or weak) but it is not useful. To balance the scale, the accuracy property will limit the *mistakes* (processes marked as failed even if they are correct) that the failure detector can make.

Definition 2 (Accuracy)**.** *The four types of accuracy are defined as follows.*

1. ***Strong Accuracy****: No process is suspected before it crashes*

2. ***Weak Accuracy****: Some correct process is never suspected*

3. ***Eventual Strong Accuracy****: There is a time after which correct processes are not suspected by any correct process*

4. ***Eventual Weak Accuracy****: There is a time after which some correct process is never suspected by any correct process*

The strong and weak accuracy are called *perpetual* properties and the eventual strong and eventual weak accuracy are called *eventual* properties.

By selecting one of the completeness properties and one of the accuracy properties, there are defined eight classes of failure detectors. Every class has its own name as it is shown in Table 2.1.

Completeness \ Accuracy	**Strong**	**Weak**	**Eventually strong**	**Eventually weak**
Strong	P Perfect	S Strong	$\Diamond P$ Eventually perfect	$\Diamond S$ Eventually strong
Weak	Q Quasiperfect	W Weak	$\Diamond Q$ Eventually quasiperfect	$\Diamond W$ Eventually weak

Table 2.1: Classes of failure detectors

2.1.3 Reducibility

To establish a *hierarchy*, Chandra and Toueg defined the concept of *reducibility* between failure detectors. Informally, one can use a failure detector $\mathcal{D}$ for implementing another failure detector $\mathcal{D}'$.

Definition 3 (Reducibility)**.** *A failure detector $\mathcal{D}'$ is reducible to failure detector $\mathcal{D}$ if there is a distributed algorithm $T_{D \to D'}$ that can transform $\mathcal{D}$ into $\mathcal{D}'$.*

The algorithm that transforms $\mathcal{D}$ into $\mathcal{D}'$ uses $\mathcal{D}$ to maintain a variable $output_p$ for every process p_i. That variable emulates the output of D' at p_i. Intuitively, failure detector $\mathcal{D}$ must provide at least as much information of failures as D' does.

If an algorithm uses failure detector $\mathcal{D}'$ for solving a problem and there exists the reduction $T_{D \to D'}$, failure detector $\mathcal{D}$ can be used instead of $\mathcal{D}'$.

We will say that $\mathcal{D} \geq \mathcal{D}'$ if $\mathcal{D}'$ is *reducible* to $\mathcal{D}$ or that $\mathcal{D}'$ is *weaker* than $\mathcal{D}$. If $\mathcal{D}' \geq \mathcal{D}$ too, it is said that $\mathcal{D}$ and $\mathcal{D}'$ are *equivalent* and write $\mathcal{D} \cong \mathcal{D}'$.

For the classes $\mathcal{C}$ and $\mathcal{C}'$ of failure detector, we have that $\mathcal{C} \geq \mathcal{C}'$ if for every failure detector $\mathcal{D}' \in \mathcal{C}'$ there is a failure detector $\mathcal{D} \in \mathcal{C}$ such that $\mathcal{D} \geq \mathcal{D}'$. As the relation of equivalence defined before, if $\mathcal{C} \geq \mathcal{C}'$ and $\mathcal{C}' \geq \mathcal{C}$, then the classes $\mathcal{C}$ and $\mathcal{C}'$ are equivalent, written as $\mathcal{C} \cong \mathcal{C}'$.

In the case in which $\mathcal{C} \geq \mathcal{C}'$ and $\mathcal{C}$ is not equivalent to $\mathcal{C}'$, we say that $\mathcal{C}'$ is *strictly weaker* than $\mathcal{C}$ and write $\mathcal{C} > \mathcal{C}'$.

A trivial reduction algorithm results by only querying the failure detector $\mathcal{D}$ and writing the current value in the output variable of every process. From the trivial reduction, the following relations are given:

$$P \geq Q, S \geq W, \diamond P \geq \diamond Q, \diamond S \geq \diamond W$$

2.1.4 Using weak completeness to simulate strong completeness

Failure detectors satisfying weak completeness can be used to simulate strong completeness. Remember that weak completeness means that every crashed process is suspected by one correct process.

The algorithm for transforming weak completeness to strong completeness is very easy. Since a correct process p_i suspects all crashed process, and the accuracy property does not change, p_i only has to send its information periodically to all the processes and that is enough for all of them to have all the failure information of the system. Algorithm 1 shows how is the transforming algorithm.

Algorithm 1 Code for process p_i

$output_p = \emptyset$

while *true* **do**
  p_i queries its local failure detector module D_i
  $suspects_i = D_i$
  send$(i, suspects_i)$ to all
end while

upon receiving $(j, suspects_j)$ from neighbor p_j
begin:
  $output_i = (output_i \cup suspects_j - \{j\})$
end

For showing that the transformation algorithm is correct, it is necessary to show that the simulated failure detector satisfies strong completeness and that it maintains its accuracy property (perpetual or eventual).

Showing strong completeness is relatively simple. For all processes p_j that fail, there is a correct process p_i that eventually suspects p_j (weak completeness). When p_i sends its suspect list to every process,every failed process p_j is marked as suspected, so every process adds p_j to its suspect list. Since p_j failed, there will be a time in which every process stops receiving messages from p_j, so they do not take out p_j from their suspect list again.

For proving that perpetual accuracy is preserved, since no process suspects p_j before time, no process sends a message with p_j in the suspect list before p_j crashes (if it does). For eventual accuracy, if p_j is correct and all the correct processes do no suspect p_j, and there is some process that suspects p_j, then it eventually fails. On the other hand, if p_j is correct, eventually every correct process p_k receives again a message from p_j, then p_k removes p_j from $output_k$. Since no correct process suspects p_j, no process sends p_j to the suspect list. Then, p_k does not add p_j to $output_k$ again.

This reduction gives place to the following relations:

$$Q \geq P, W \geq S, \diamond Q \geq \diamond P, \diamond W \geq \diamond S$$

Corollary 1. *$Q \cong P$, $W \cong S$, $\diamond Q \cong \diamond P$ and $\diamond W \cong \diamond S$*

And as a consequence of the given reductions is the Corollary 1, this means that the given reduction algorithms collapses the eight failure detector classes into four, but one natural question is *What is the relation between the four collapsed classes?* There are some classes that are incomparable between them as is shown in [12].

Theorem 2. *$P > S$, $\diamond P > \diamond S$, $P > \diamond P$, $S > \diamond S$, $P > \diamond S$ and, S and $\diamond P$ are incomparable.*

2.2 Using failure detectors for solving problems

Algorithms for solving consensus and failure detectors that are not in the hierarchy of Chandra and Toueg that were introduced for solving different problems in the distributed computing area are shown in this section.

For the following algorithms it is assumed a *Reliable Broadcast* communication primitive (otherwise, in a purely asynchronous system failure detectors would not be able to help solving consensus) as it is defined in [12] that satisfies the following properties:

- *Validity*: If a correct process R-broadcast a message m then it eventually R-delivers m
- *Agreement*: If a correct process R-delivers a message m then all correct process eventually R-deliver m

- *Uniform integrity*: For any message m every process R-delivers m at most once, and only if m was previously R-broadcast by some process.

The implementation of Reliable Broadcast for asynchronous systems that is used in the next algorithms can be found in [12].

2.2.1 S for solving consensus

In this section a particular implementation of the consensus using a failure detector of class S is presented. This implementation is an instance of the shown in [12].

Algorithm 2 has three phases. In the first phase, processes exchange information. Initially, every process p_i sends its value to every process and in the subsequent rounds, p_i sends the values it learned in the previous rounds. When every process p_i waits for the value of the other processes, let us say p_j, p_i waits for the value of p_j as long as the failure detector does not suspect p_j. In the second phase, every process p_i sends its estimated vector to every process p_j. Again, p_j waits for the message of p_i as long as p_j does not suspect p_i. Then, processes compare their estimated vectors and in the third phase, processes decide.

The algorithm satisfies termination. It can only be blocked in the *wait* statements, but remember that S satisfies strong completeness, meaning that eventually every correct process p_i eventually suspects every crashed process. So if p_i waits for a message from p_j and p_j failed, p_i eventually suspects p_j and does not wait forever. It satisfies validity because in every phase, it stores in the estimated vector a value proposed by a process or $\perp$.

Since S satisfies weak accuracy, there will be a correct process c that is never suspected by any process, meaning that at least the message of c is received in phase 1 and 2. At the end of phase 2, the vector of estimated values must be the same as the vector of the process c, which guarantees that at the end every process decides the same value (agreement).

2.2.2 $\Diamond S$ for solving consensus

In this section it is shown how to solve the consensus using an eventually strong failure detector $\Diamond S$. As it is something to note, the implementation using S is very different from the one that is shown in this section. It is needed to assume that $f < \lceil \frac{n}{2} \rceil$, i.e. a majority of processes is correct.

For this algorithm the *coordinator paradigm* is used . In every round, there will be a coordinator that tries to gather the information, lock a value and then decide.

Algorithm 3 works in asynchronous rounds. Every round has four phases and at the beginning of the round a coordinator c is calculated. In the first phase, every process sends its estimate to the coordinator and it waits for a majority of values received. In the second phase, the coordinator estimates a new value and it is communicated to every process p_j. In the third phase, every process p_j waits for the estimated of the coordinator as long as p_j does not suspect c. If p_j receives the estimated value of c, p_j sends an acknowledge message to c an adopts that value as a new estimate. Otherwise,

Algorithm 2 Code for process p_i

Variables
$V[] = \emptyset$
$aux[] = V$
$msgs = []$
$V[i] = v_i$

for $r = 1$ to $n - 1$ **do** ▷ Asynchronous Rounds
  $send(r, aux, i)$ to all neighbors
  wait until $\forall j : received(r_j, aux_j, j)$ or $j \in \mathcal{D}_i$ ▷ Wait while p_j is not suspected
  $msgs[r] = \{(r, aux_j, j) | received(r, aux_j, j)\}$ ▷ All the messages of round r
  $aux[] = \emptyset$
  for $k = 1$ to n **do** ▷ Collects values of other processes
    if $V[k] = \emptyset$ and $\exists (r, aux_j, j) \in msgs[r]$ such that $aux_j[k] \neq \emptyset$ **then**
      $V[k] = aux_j[k]$
      $aux[k] = aux_j[k]$
    end if
  end for
end for

$send(V)$ to all neighbors
wait until $\forall j : received(V_j)$ or $j \in \mathcal{D}_i$ ▷ Wait while p_j is not suspected
$last\_msgs = \{V_j | received(V_j)\}$
for $k = 1$ to n **do**
  if $\exists V_j \in last\_msgs$ with $V_j[k] = \emptyset$ **then**
    $V[k] = \emptyset$
  end if
end for

$decide$(first nonempty entry of V)

sends a message of non-acknowledgment. In the fourth phase, the coordinator waits for a majority of acknowledgment messages. If that is the case, the coordinator communicates to every processes the decision.

The coordinator is not blocked since it is assumed that $f < \lceil \frac{n}{2} \rceil$, so eventually receives the values of a majority of processes. Since $\Diamond S$ satisfies strong completeness, eventually every crashed process is suspected by every correct processes and the wait statement of phase 3 does not block the algorithm.

$\Diamond S$ satisfies eventual weak accuracy, meaning that eventually there is a process $\rfloor$ that is not suspected by any correct process. Eventually, there will be a round coordinated by $\rfloor$, so in phase 3 all the correct processes wait for the estimated value of $\rfloor$ since none of them suspect $\rfloor$. Finally, the correct processes send its acknowledgment to the coordinator.

Algorithm 3 Code for process p_i

Variables

$estimate = v$ ▷ Estimated decision value. Initialized with self-value.
$msgs = []$ ▷ Messages of every round are stored here
$state = undecided$ ▷ Current state of the decision
$r = 0$ ▷ Current round number
$last\_round = 0$ ▷ Last round number in which p_i updated its $estimate$
$c = 0$ ▷ Current coordinator
while $state = undecided$ **do**
  $r = r + 1$
  $c = (r \ mod \ n) + 1$ ▷ Calculate the current coordinator
  Phase 1
  $send(i, r, estimate, last\_round)$ to coordinator c ▷ Send the estimated value to the coordinator
  Phase 2
  if $i = c$ **then** ▷ If p_i is the coordinator
    wait until $\lceil \frac{n+1}{2} \rceil$ messages $(j, r, estimate_j, last\_round_j)$ are received ▷ Wait for the estimated of a majority of processes
    $msgs[r] = \{m = (j, r, estimate_j, last\_round_j) | p_i$ received m from $p_j\}$ ▷ All the messages of round r
    $t = max\{last\_round_j | (j, r, estimate_j, last\_round_j) \in msgs[r]\}$ ▷ The most recent round in which p_j updated its $estimate_j$ from the messages received in round r
    $estimate = \{estimate_j | (j, r, estimate_j, t) \in msgs[r]\}$
    $send(i, r, estimate_i)$ to all neighbors
  end if
  Phase 3
  wait until $received(c, r, estimate_c)$ from c or $c \in \mathcal{D}_i$ ▷ Wait for the message of the coordinator while it is not suspected
  if $received(c, r, estimate_i)$ from c **then**
    $estimate = estimate_{c_i}$
    $last\_round = r$
    $send(i, r, ack)$ to c
  else
    $send(i, r, nack)$ to c
  end if
  Phase 4
  if $i = c$ **then**
    wait until $\lceil \frac{n+1}{2} \rceil$ messages (j, r, ack) o $(j, r, nack)$
    if $\lceil \frac{n+1}{2} \rceil$ messages (j, r, ack) are received from processes j **then**
      $R - broadcast(i, r, estimate, decide)$
    end if
  end if
  when R deliver $(j, r_j, estimate_j, decide)$
  if $state = undecided$ **then**
    $decide(estimate_j)$
    $state = decided$
  end if
end while

2.2.3 The weakest failure detector for solving consensus

The weakest failure detector of the hierarchy proposed by Chandra and Toueg for solving Consensus in any system with $n > 2f$ is of class $\Diamond W$, the eventually weak failure detector [11]. They proved that $\Diamond W$ can be reduced to every failure detector $\mathcal{D}$ that can be used to solve the consensus, i.e. $\Diamond W$ is weaker than any failure detector that can be used to solve Consensus. The idea for proving that, is that any failure detector that can be used for solving the consensus must provide at least as much information as $\Diamond W$ and then show how to transform any failure detector to $\Diamond W$.

A very superficial look to the proof consists of the following. It is assumed that there is an algorithm $\mathcal{A}$ that implements consensus using a failure detector $\mathcal{D}$. Then, it is aims at giving a transformation algorithm T for implementing the failure detector of class Ω (we talk about it in section 2.2.4) in which eventually, every correct process must elect a correct process as a leader. And finally, there is a reduction that transforms Ω into $\Diamond W$. The proof in [11] is a very extensive and technical proof. In [25] is given a less technical and friendly explanation of the proof and in [16] is given a simple transformation for reducing Ω to $\Diamond W$.

2.2.4 Other classes of failure detectors

For solving different problems in the distributed computing area, some failure detectors were designed that are not in the original hierarchy of Chandra and Toueg. Instead, new classes of failure detectors have been proposed for the same model of computation. In this section some of these failure detectors are described.

Leader failure detector

The *Leader failure detector* Ω [36, 12] is a weaker version of the *Leader election problem* that is defined as follows. Each process p_i has a local variable $leader_i$, and it is required that all the local variables $leader_i$, once changed from its initial values, forever contain the same identity, which is the identity of one of the processes. A classical way to elect a leader consists in selecting the process with the smallest (or largest) identity[1].

Leader election has been intensively investigated in asynchronous failure-free systems where the processes communicate by message-passing. If processes may crash, the system is fully asynchronous, and the elected leader must be a process that does not crash, leader election cannot be solved [47]. Not only the system must no longer be fully asynchronous, but the leader election problem must be weakened to the *eventual leader election problem*. This problem emerges naturally from the $\Diamond W$ failure detector.

The Ω failure detector was proposed in [11] for showing that $\Diamond W$ is the weakest failure detector that can be used for solving the consensus. They claimed that Ω is at least as strong as $\Diamond W$, so every failure detector $\mathcal{D}$ that is as as strong as Ω is as strong as $\Diamond W$.

[1] A survey on election algorithms in failure-free message-passing systems appears in Chapter 4 of [46]. The aim is to elect a leader as soon as possible, and with as few messages as possible, and it can be done on a ring with $1.271\ n\ \mathsf{log}(n) + O(n)$ messages [31, 6].

Definition 4. *The leader failure detector satisfies the following properties [44]:*

- Validity: *Each invocation of* Ω *returns a process name*
- Eventual leadership: *Exists time* t*, such that for all time* $t' > t$ *and for some correct process* p_i*, every invocation of* Ω *for every correct process returns* p_i

Quorum failure detector

The *quorum* failure detector, denoted as Σ, was introduced in [19]. In [20] is showed that Σ is the weakest failure detector that can be used for implementing an atomic register in asynchronous message-passing systems. It is important to mention that a register can be implemented if a majority of processes is correct [7] without using any failure detector. But using the failure detector implements the atomic register without the majority assumption.

The output of failure detector Σ at process p_i at any time t, consists of a set of processes that are said to be *trusted* by p_i.

Definition 5. *The quorum failure detector satisfies the following properties [19]:*

- Intersection*: Given any two lists of trusted processes at any time and by different processes, at least one process belongs to both lists*
- Completeness*: Eventually no crashed process is ever trusted by any correct process*

From the intersection and completeness properties the *accuracy* can be derived as follows: Every set of correct processes contains at least one correct processes.

Anonymously perfect failure detector

The anonymously perfect failure detector, denoted as $?\mathcal{P}$ was introduced in [28] for solving the Non-Blocking Atomic Commit [50] in which eventually, every correct process should agree in a common decision for a transaction: *commit* or *abort*.

The failure detector class $?\mathcal{P}$ gives information about failures but without indicating which process failed, only that there has been a failure. A failure is detected if only if it actually happened.

Definition 6. *The anonymously failure detector satisfies the following properties [28]:*

- Anonymous Completeness*: If some process crashes, eventually every correct process permanently detects a crash*
- Anonymous Accuracy*: No crash is detected unless some process crashes*

Heartbeat failure detector

The heartbeat failure detector ($\mathcal{HB}$) was proposed in [1] for solving the quiescent reliable communication problem (i.e. algorithms that eventually stops sending messages). This class of failure detector outputs a vector of counters, with one entry for every process. If a process crashes, the counter does not increases, otherwise, it is not bounded.

Definition 7. *The heartbeat failure detector satisfies the following properties [1]:*

- $\mathcal{HB}$-Completeness*: For every correct process p_i and for every crashed process p_j neighbor of p_i, the heartbeat counter of p_j is bounded in p_i*
- $\mathcal{HB}$-Accuracy*: For every correct process p_i, the heartbeat counter is non-decreasing and for every correct neighbor p_j of p_i, the hearbeat counter of p_j in p_i is unbounded*

2.3 Algorithm design for failure detectors

Since the beginning of this chapter, we have considered the failure detectors as a black-box that satisfies certain properties for solving problems, or for proving which is the weaker failure detector that can solve a problem, but we have not focused on a particular network model or given an actual algorithm for implementing a failure detector. In [12] it was proposed the first implementation of a failure detector of class $\Diamond P$ which consists in encapsulating techniques already known such as timeouts.

For designing an algorithm for failure detection, it is needed to consider different aspects of the network such as the communication model (faulty links), level of synchrony of the system (partially synchronous, synchronous or asynchronous), the topology of the network (complete or arbitrary) and which kind of failure we are coping (crash, omission, byzantine). Once we have establish which kind of network we need to assume and which class of failure detector we want to implement, we need to know whether it is possible to implement that kind of failure detector or a weaker one. For example, it has been shown in [39] that the failure detectors that satisfy any perpetual accuracy cannot be implemented in a partially synchronous system with failures.

Some of the network assumptions may cause that no implementation of a failure detector can satisfy completeness and accuracy, for example, if we consider partitionable networks, neither the completeness nor accuracy defined in [12] can be satisfied since processes in distinct connected components (partitions) cannot distinguish if a processes failed or it is in a different partition.

As we showed in section 2.2.4, failure detectors not in the hierarchy of Chandra and Toueg have been proposed for solving different problems for the same model of computation, but what if we need to change that model? Then we need to propose an extension of the completeness and accuracy properties.

In the following sections are shown some models of computation and their different extensions of the completeness and accuracy properties.

2.3.1 Partitionable networks

If it is considered a network of arbitrary topology with no further assumptions, due to failures the network may be broken into pieces called *partitions*. As was explained before, under this assumption it is no possible to satisfy the completeness and accuracy defined by Chandra and Toueg, so it is needed to extend the properties. In [2] it is defined the extension of the completeness and accuracy properties for partitionable networks.

Definition 8. *An eventually strong failure detector $\Diamond S$ for partitionable networks satisfies the following properties [2]:*

- Strong completeness*: For every process p_j such that p_j has crashed or is not in the same partition of p_i, eventually p_i suspects p_j*
- Eventual weak accuracy*: For every partition P there is a correct process p_j that is not suspected by any correct process p_i in the same partition*

2.3.2 Networks with unknown membership

Another interesting and realistic model are the networks with unknown membership, i.e. networks in which the number and the name of the participants are not known by any participant in the network. In [33] it is showed that no failure detector of the hierarchy of Chandra and Toueg can be implemented in a network with unknown membership. This is because in every algorithm that runs in a system where there is at least one process that has not knowledge of at least some other process, weak completeness cannot be satisfied.

Again, it is needed to extend the properties for networks with unknown membership. In this case, every process needs to communicate its name to other processes. We will say that a correct process p_i *knows* process p_j if p_j is in the list of known processes of p_i. A process p_j is *known* if there is some correct process p_i that knows p_j. In [27] is proposed an implementation of $\Diamond S$ for networks with unknown membership.

Definition 9. *An eventually strong failure detector $\Diamond S$ for networks with unknown membership satisfies the following properties [2]:*

- Strong completeness*: Eventually, for every faulty and known process p_j, it is suspected by every known correct process p_i*
- Eventual weak accuracy*: Eventually, there is a correct and known process p_j that is not suspected by any correct and known process p_i*

2.4 Related work to $\Diamond P$

Failure detectors were introduced by Chandra and Toueg [12] and are classified by completeness and accuracy properties, for non-partitionable networks. Some classes

of failure detectors of this hierarchy output a list of processes that are suspected to have failed, but there are many other classes of failure detectors. In this thesis it is presented an implementation of $\Diamond P$ for complete networks with crash failures and unreliable channels that eventually delivers a message.

Aguilera, Chen and Toueg [1] propose the *Heartbeat* failure detector, not in the original hierarchy of Chandra and Toueg. The Heartbeat failure detector does not use timeouts, instead they use an unbounded counter that increases with every message received by the process. The failure detector module of every process does not output a suspect list, instead the output is a vector with unbounded counters. The network model assumes that channels connecting processes may fail by dropping messages or by disconnecting processes, producing partitions of the network into maximal connected components. The completeness and accuracy properties were adapted for partitionable networks [2].

In [3], it was given an implementation of $\Diamond P$ for complete networks with message losses and n participating processes, where only n bidirectional links are required to be eventually timely.

In [39] the implementability of different classes of failure detectors in several models of partial synchrony is studied. It is shown that no failure detector with perpetual accuracy (namely, P, Q, S, and W) can be implemented in these models in systems with even a single failure. Also, in these models of partial synchrony, a majority of correct processes is necessary to implement a failure detector of the class θ proposed by Aguilera et al. Finally, a family of distributed algorithms is presented that implements the four classes of unreliable failure detectors with eventual accuracy (namely, $\Diamond P$, $\Diamond Q$, $\Diamond S$, and $\Diamond W$). The algorithms are based on a logical ring arrangement of the processes, to define the monitoring and failure information propagation pattern.

Another algorithm for $\Diamond P$ for arbitrarily connected networks is presented in [32]. This algorithm uses unbounded heartbeat counters and timeouts. In this algorithm, every process does not need to know the name of all processes in the network, but it needs to know which processes are its neighbors and more importantly, it needs to know the bound on the variability of the delays on the communication between neighbors.

Algorithms that implement failure detectors in partially synchronous systems are presented in [36]. A $\Diamond P$ optimal implementation in terms of the number of bidirectional links is described. Observe that, if (uni)directional links are considered, $\Diamond P$ can be implemented even if only n directional links carry messages forever [40].

The *Average Delayed/Dropped* (ADD) channels were introduced in [48]. They provide a very weak communication model. An algorithm for implementing $\Diamond P$ on a complete network connected by ADD channels is proposed in the same paper. An implementation of the eventually perfect failure detector in an arbitrary, partitionable network composed of ADD channels using messages of size $O((n+1)!)$ bits is described in [34]. In [49] there where established necessary and sufficient conditions for crash-quiescent failure detection in a system with ADD channels.

2.5 Related work to Ω

Many algorithms electing an eventual leader in crash-prone partially synchronous systems have been proposed. Surveys of such algorithms are presented in [45, Chapter 17] when communication is through a shared memory, and in [47, Chapter 18] when communication is through reliable message-passing.

A communication optimal implementation of $\Diamond S$ (recall that $\diamond S \cong \diamond W \cong \Omega$) is presented in [38], where the network is complete and every process is connected by a reliable communication channel. It uses messages of size $O(log\ n)$ and it is focused on the number of processes that the leader communicates with. It proposes also a measure to evaluate the efficiency of algorithms implementing failure detectors called the *monitoring degree* which is the number of processes that are monitored at some time.

In [37] there are proposed algorithms for every class of the original hierarchy of Chandra and Toueg. It is assumed that the network is complete, every channel is reliable and processes are arranged in a ring. The aim of those algorithms is to send messages in at most $2n$ links.

In [4] there are proposed different levels of communication reliability and is its showed that in systems with only some timely channels and a complete network it is necessary that correct processes send messages forever even with just at most one process crash. More precisely, it is showed that Ω can be implemented in a complete network with only one processes having timely output channels. Despite being a complete network, there might be the case in which a pair of processes cannot communicate with each other (for arbitrary delays) and the easy way to elect the leader (selecting the process with minimum id) cannot be used. Instead, the less suspicious process is elected. An Ω implementation in the previous described model that is *communication-efficient* i.e. eventually only one process sends message is given. Another implementation in networks whose all links are *fair* is given. A fair link may lose an infinite number of messages but if a message is repeatedly sent, then it is eventually delivered [2].

An algorithm for implementing Ω in networks with unknown membership is presented in [33]. This algorithm works in a complete network and every process needs to communicate its name to every neighbor using a broadcast protocol. In [23] it is presented an implementation of Ω for the case of the crash-recovery model in which processes can crash and then recover infinitely many times and channels can lose messages arbitrarily. It assumes the existence of a *core* that remains connected, it is partially synchronous and eventually all its processes are correct.

The case for dynamic systems is addressed in [41], and the case where the underlying synchrony assumptions may change with time is addressed in [5]. Stabilizing leader election in crash-prone synchronous systems is investigated in [17].

Chapter 3

Formal model

A failure detector cannot be implemented in a purely asynchronous system since it would contradict the FLP impossibility result. In [21] are described different types of partial synchrony and defined what is the least amount of synchrony sufficient to solve consensus.

In real life scenarios, despite having most of the time reliable communication, we do not have any timing guarantee which means that the worst case scenario (arbitrary delays and failures) can actually happen. For this work, we use the ADD model first proposed by Sastry et al in [48], which models a scenario as the described above.

This chapter introduces the formal model in which the algorithms of Chapters 4 and 5 were designed.

3.1 Processes

The *system* consists of a finite set of processes $\Pi = \{p_1, p_2, ..., p_n\}$. Every process p_i has an identity, and without loss of generality we consider that the identity of p_i is its index i. As there is no ambiguity, we use indifferently p_i or i to denote the same process.

Every process p_i has also a read-only local clock $\mathsf{clock}_i()$, which is assumed to generate ticks not necessarily at constant rates, but with bounded drift. Local clocks need not to be synchronized and are used only to implement timers. To simplify the presentation, it is assumed that local computations have zero duration.

For notational simplicity, in the explanation and the proofs, we assume the existence of an external reference clock which remains always unknown to the processes. The range of its ticks is the set of natural numbers. It allows to associate consistent dates with events generated by the algorithm.

3.2 Communication graph

The communication graph is represented by a directed graph $G = (\Pi, E)$, where an edge $(p_i, p_j) \in E$ means that there is a unidirectional channel that allows the process p_i to send messages to p_j. A bidirectional channel can be represented by two unidirectional

channels, possibly with different timing assumptions. Thus, each process p_i has a set of input channels and a set of output channels.

The graph connectivity requirement on the communication graph G depends on the problem to be solved. It will be stated in every chapter, depending on the presented algorithm.

3.3 Distributed algorithms

A *distributed algorithm* is a collection of local algorithms $A_1, A_2, \ldots, A_n$. Every process $p_i \in \Pi$ follows the specification of A_i. An *execution* is an infinite sequence $\alpha = C_0, s_0, C_1, s_1, \ldots$ with C_0 being the initial configuration of the system and C_{i+1} being the resulting configuration of applying the event s_i to the configuration C_i. Every *event* s_i corresponds to the specification of A_j, which can be sending or receiving a message or a local computation.

Processes may fail only by crashing. Given an execution α, a process p_i is said to crash at time t if p_i does not perform any event in α after time t and $\mathsf{crashed}(\alpha, t)$ is the set of all processes that have crashed by time t. A process p_i is correct at time t if p_i has not crashed by time t and $\mathsf{correct}(\alpha, t) = \Pi - \mathsf{crashed}(\alpha, t)$.

If a process $p \in \mathsf{crashed}(\alpha, t)$ at some t, we say that p_i is a *faulty* process in α. If a process $p \in \mathsf{correct}(\alpha, t)$ for all t, we say that p_i is a *correct* process in α. Graph G at time t is defined as $G(t) = (\mathsf{correct}(\alpha, t), E')$ with $E' = \{(u, v) | (u, v) \in E$ and $u, v \in \mathsf{correct}(\alpha, t)\}$. A *partition* P is a maximal strongly connected component of G.

3.4 Initial knowledge of a process

All processes know the name of every process in the network. This assumption is not trivial, as it has been shown in [33] that without this assumption, it is not possible to implement a failure detector class even in a fully synchronous system with reliable links. However, we discuss extensions of failure detector definitions where this assumption is no longer true (as discussed in the Introduction).

3.5 *ADD* Channels

A directed channel (p_i, p_j) satisfies the *ADD property* if there are two constants K and D (unknown to the processes[1]) such that

- for every K consecutive messages sent by p_i to p_j, at least one is delivered to p_j within D time units after it has been sent. The other messages from p_i to p_j can be lost or experience arbitrary delays

Each directed channel can have its own pair (K, D). To simplify the presentation, and without loss of generality, we assume that $K = max\{K_{i,j}\}$ and $D = max\{D_{i,j}\}$.

[1] Always unknown, as the global time, is also never known by the processes.

3.5.1 The $\Diamond$ADD property

The eventual ADD property, states that the ADD property is satisfied only after an unknown but finite period of time. Hence this weakened property allows the system to experience an initial anarchy period during which the behavior of the channels is arbitrary.

Chapter 4

$\Diamond P$ implementations

This chapter presents implementations of $\Diamond P$ using messages of size $O(n \log n)$, in arbitrarily connected network of ADD channels and networks with unknown membership. We are inspired by the networking technique of *time-to-live* (TTL) values[1] to design a very flexible, novel failure detector and then extend it to more dynamic networks.

TTL's are commonly used for limiting the lifetime of packets in a network or the number of hops a packet can take [35, 51]. We use this idea in our $\Diamond P$ implementation, to solve the challenge of sending messages of small size. We implement the following analogy. Each process p_i emits heartbeats at a certain frequency, and with maximal *intensity*. The intensity is encoded by an integer value, that works similar to a TTL value. These heartbeats are propagated by echoing through the network, perhaps varying the frequency, due to the K, D bounds of each channel they traverse. They lose intensity as they go farther away from its origin p_i. If p_i crashes its heartbeats eventually fade out. But if p_i remains alive, all processes (in the connected component of p_i) keep on hearing from p_i's heartbeats, although perhaps with low intensity, if a process is far away.

In more detail (but still very roughly), each process p_i periodically sends a set of up to n elements. This set indicates that p_i is alive, and summarizes what p_i knows about other processes. Each element is a pair (p_j, m) consisting a process identifier p_j and an integer value m between 1 and $n-1$ that its called *hopbound*. Thus, a maximum of $n \log n$ bits in total. The m component of a pair contains the current intensity of the heartbeat of process p_j. For itself, process p_i sends a pair with intensity $n-1$ because this value is strong enough to guarantee that the heartbeats of p_i can reach all the nodes in the network.

Whenever p_i receives a set with hopbound values from one of its neighbors, it must update its own knowledge (in a somewhat delicate way), and send the new set to its neighbors, making sure that the hopbound values received are forwarded with an intensity decremented by one. In particular, the node takes the largest hopbound value it learns about another process p_j, and resends it. But it cannot resend it only once,

[1] Time to live (TTL) is a mechanism that limits the lifetime of data in a computer or network. Once the prescribed event count or timespan has elapsed, data is discarded or revalidated. In computer networking, TTL prevents a data packet from circulating indefinitely. Source: Wikipedia.

because an ADD channel could lose it. Thus, it repeatedly resends it, until due to some suspicion, it starts to accept lower hopbound values for that node, p_j. The first challenge is that if a process p_j crashes, eventually its hopbound values should disappear and everyone must suspect it. The opposite challenge is to make sure that if p_j does not crash, eventually nobody suspects it. For this, a process p_i that learns a hopbound value from p_j must resend it repeatedly to make sure the information is not lost by an ADD channel, since particular messages on an ADD channel may be lost. Our algorithm is carefully tuned to balance these two challenges, and to make sure p_i stops resending the hopbound of p_j if it does not eventually get fresh information from which it can deduce that p_j remains alive.

4.1 In partitionable networks

For this implementation we consider a partitionable network connected with ADD channels. Every process knows n and the name of every participant in the network.

4.1.1 $\Diamond P$ specification

We will say that a process p_i *suspects* process p_j if p_j is in the failure list of p_i. A *partition* P is a maximal strongly connected component of G.

The oracle $\Diamond P$ is formally specified as follows. For each execution α and every correct process p_i in α, there is a time t^a such that for every $t > t^a$ the failure detector satisfies:

- **Strong completeness**: For every process p_j such that it is crashed or it is not in the same partition of p_i in $G(t)$, p_i suspects p_j at time t
- **Eventual strong accuracy**: For every correct process p_j in the same partition of p_i in $G(t)$, p_i does not suspect p_j at time t

4.1.2 Description of the algorithm

In this section we describe our algorithm. Algorithm 8 uses the standard technique of heartbeats and timeouts, i.e. a process sends periodically a message to its neighbors. Its neighbors estimate the time of arrival of a new message, managing timeouts. We extend these traditional ideas, with time-to-live values.

Algorithm 8 is composed of three main modules. The first module (line 7) prepares and sends periodically heartbeat messages to neighbors. The second module deals with the reception of messages from neighbors (line 19), and timeouts are estimated. The third module deals with the expiration of the timer that the algorithm uses. When the timeout of a process expires, that process is suspected (line 35).

In what follows a subscript in the name of a variable is used for denoting which process it belongs to; for example the $timeout_i[j]$ corresponds to p_j's timeout entry in p_i and $hopbound_i$ will denote a hopbound value of p_i.

Alive messages. Every process p_i sends an alive message to all its neighbors every T units of time (as usual, T can be adjusted to save bandwidth or improve response time, as desired). This alive message contains a set, which is stored in the local variable bag_i and it is called *bag*. When Algorithm 8 starts, the bag contains only a pair $(i, n-k)$, where the $n-k$ variable is an integer called *hopbound* (line 9) and in this particular case $k = 1$ since is the origin of the message. When a process p_i receives pairs from its neighbors, it gathers all the information to create a new set (line 12) and sends this new set to all its neighbors (line 15).

Reception of the alive messages. When process p_i receives an alive message from a neighbor p_j, the message contains a set with several pairs, potentially a pair for every process in the network. Process p_i only takes information from a pair labeled with p_j sent by p_j. By only taking information about neighbors directly from neighbors we are giving priority to the topology graph, namely, at some point, a process p_j at distance k from p_i eventually will have stored permanently the pair $(i, n-k)$.

When a set is delivered, p_i takes the information from pairs (ℓ, m) such that p_ℓ is not a neighbor of p_i or $\ell = j$, since neighbor p_j sends information of itself directly (line 21). Process p_i stores the received hopbound of process p_j in $hopbound_i[j]$.

Timer estimation. When p_i receives an alive message from a neighbor p_j, it saves its hopbound value (which will be always $n-1$) and estimates how much time will take to receive another message from p_j. This estimation is in $timeout_i[j]$. If this timeout expires, p_i suspects p_j and sets $suspect_i[j]$ to true (line 35). If p_i receives a message of p_j and $suspect_i[j]$ is true, probably p_j was wrongly suspected by p_i and the timeout value was estimated to be too small, so p_i stops suspecting p_j by setting $suspect_i[j]$ to false and incrementing the $timeout_i[j]$ (line 26). Process p_i does not suspect process p_j if p_i receives periodically heartbeats from p_j on time, (namely, before the timeout expires).

The same idea is used for processes that are not neighbors of p_i, but in this case their timeout is estimated in relation to their hopbound value, as it is explained below.

The algorithm uses the function *estimateTimeout* (line 32) to increase the timeout each time a false suspicion is detected. For correctness, we need to assume only that the function increments the timeout. For performance, one may tune this increment. Here we use the constant 2 to reach a correct timeout exponentially fast.

Time-to-live values. This is the strategy used for preventing that a pair labeled with a crashed process remains forever in the network. Every time a process p_i sends an alive message, it first decrements by one the hopbound value of every process. A pair $(j, hopbound_i[j])$ will only be added to the set of pairs the process will send, if $hopbound_i[j] - 1 > 0$ (line 12). On the other hand, since the longest path in the network may be of length $n-1$, a process p_i adds to its own set the pair $(i, n-1)$, to guarantee that every process is informed that it is alive, by receiving a pair $(i, n-k)$ with $k > 1$.

Intuitively, process p_i should store the hopbound of a process p_j every time it receives the pair (ℓ, m) in the set of any neighbor p_j. Because channels are unreliable, and they have different latencies, it may be the case that a process p_k wrongly suspects a process p_j. Hence, a process always keeps the $max(hopbound_i[\ell], m)$.

We need to avoid the situation that if p_j fails, the other processes will keep receiving messages labeled with p_j forever. To solve this problem, the maximum hopbound of p_j received by process p_i sent by p_j is stored (first condition in line 22). If this value is not received again before the $timeout_i[j]$ expires, process p_j is suspected by p_i and p_i stores any other hopbound received, which must be a smaller one (second condition in line 22). Thus, if p_j fails, eventually all the pairs labeled with j will fade away. Hence, eventually every correct process suspects p_j.

Each process p_i manages the following local variables:

- $in\_neighbors_i$ (resp., $out\_neighbors_i$) is a (constant) set containing the identities of the processes p_j such that there is channel from p_j to p_i (resp., there is channel from p_i to p_j).
- T is an integer, time between successive heartbeats.
- n is an integer, number of processes in the network.
- $clock_i()$ is the local clock of process p_i.
- $lastAlive_i$ is an array of clock times, $lastAlive_i[j]$ stores the last time that p_i received a pair from p_j.
- $timeout_i$ is an array of integers of estimated timeouts for all processes.
- $suspect_i$ is an array of booleans for suspecting processes.
- bag_i array for storing the ordered pairs (process, hopbound) for sending to neighbors.

4.1.3 Correctness proof

To prove the correctness of the algorithm, we need to show that the implementation above satisfies **strong completeness** and **eventual strong accuracy**. In what follows, we consider an infinite execution α of Algorithm 8.

First, we prove some preliminary lemmas. The following simple lemma is similar to results proved in [48] and [34].

Lemma 3. *Let p_i and p_j be two correct neighboring processes. Then, there is an upper bound $\Delta = K \times T + D$ on the time between the consecutive reception at p_j of two alive messages from p_i.*

Proof The channel delivers correctly one message for every K consecutive messages sent by p_i to p_j with delay at most D. This means that at most $K - 1$ messages are lost between two messages delivered consecutively to p_j. Recall that p_i sends a message every T ticks in line 7. Since we are assuming clocks run at constant speed, the maximum time between the consecutive reception of two messages from p_i in p_j is $K \times T + D = \Delta$. $\square_{Lemma\ 3}$

Algorithm 4 Code for process p_i

Constants

1: $neighbors, T, n$

Variables

2: $clock_i() = 1$
3: **for** each j in Π **do**
4: $\quad lastAlive_i[j] = 0$; $timeout_i[j] = T$; $suspect_i[j] = false$;
5: **end for**
6: $bag_i \leftarrow \{\emptyset\}$

7: **every** T units of time of $clock()_i$ **do**
8: **begin:**
9: $\quad bag_i \leftarrow \{(i, n-1)\}$
10: $\quad$ **for** each $j \in \Pi - \{i\}$ **do**
11: $\quad\quad$ **if** ($suspect_i[j] = false$ and $hopbound_i[j] > 1$) **then**
12: $\quad\quad\quad bag_i \leftarrow bag_i \cup \{(j, hopbound_i[j] - 1)\}$
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: $\quad$ **for** each $j \in out\_neighbors_i$ **do**
16: $\quad\quad$ send ALIVE(bag_i) to p_j
17: $\quad$ **end for**
18: **end**

19: **when** ALIVE(bag) **is received** % from p_j in $in\_neighbors_i$
20: **begin:**
21: $\quad$ **for** each $(\ell, m) \in bag$ such that $\ell \notin neighbors \setminus \{j\}$ **do**
22: $\quad\quad$ **if** ($hopbound_i[\ell] \geq m$) or ($suspect_i[\ell] = true$) **then**
23: $\quad\quad\quad hopbound_i[\ell] \leftarrow m$
24: $\quad\quad\quad$ **if** ($suspect_i[\ell] = true$) **then**
25: $\quad\quad\quad\quad suspect_i[\ell] \leftarrow false$
26: $\quad\quad\quad\quad$ ESTIMATETIMEOUT(j)
27: $\quad\quad\quad$ **end if**
28: $\quad\quad\quad lastAlive_i[\ell] \leftarrow clock()$
29: $\quad\quad$ **end if**
30: $\quad$ **end for**
31: **end**

32: **function** ESTIMATETIMEOUT(j)
33: $\quad timeout_i[j] \leftarrow 2 \cdot timeout_i[j]$
34: **end function**

35: **when** $timeout_i[j] = clock_i() - lastAlive_i[j]$
36: **begin:**
37: $\quad suspect_i[j] \leftarrow true$
38: **end**

Given two processes $p_i, p_j \in \Pi$ connected by a channel, in what follows we call Δ the bound given by Lemma 3.

Observation 4. *Given two correct neighboring processes p_i and p_j, after an interval of time at most Δ, process p_j receives the first alive message from p_i.*

G at time t is defined as $G(t) = (\mathsf{correct}(\alpha, t), E')$ with $E' = \{(u, v) | (u, v) \in E$ and $u, v \in \mathsf{correct}(\alpha, t)\}$.

Let t^f be the earliest time when all the failures in α have occurred.

Lemma 5. *The following properties hold:*

1. *$G(t) = G(t')$ for all $t, t' \geq t^f$,*
2. *There is a time $t^\epsilon \geq t^f$ after which the last message from the set of crashed processes is delivered to a correct process.*

Proof The proof of part 1 is as follows. At time t^f all the failures have occurred, then it is true that $\mathsf{crashed}(\alpha, t) = \mathsf{crashed}(\alpha, t')$ for all $t, t' \geq t^f$. Since $\mathsf{correct}(\alpha, t) = \Pi - \mathsf{crashed}(\alpha, t)$, then by the definition of $G(t)$ it is true that $G(t) = G(t')$ for all $t, t' \leq t^f$.

The proof of part 2 is as follows. Every faulty process sends a finite number of messages before crashing. Then, by the properties of the ADD channel, these messages are lost, delivered or experience arbitrary delays. So there exists a time $t^\epsilon \geq t^f$ after which the last message sent by the set of faulty process is delivered.

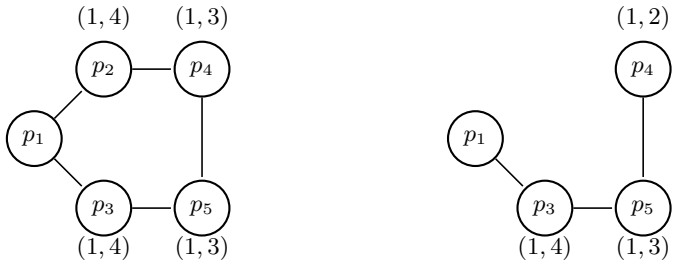
$\square_{Lemma\ 5}$

Part 1 of Lemma 5 shows that there is a time after which the topology of the communication graph does not change, once all the failures have occurred. We will call $G(t^\epsilon)$ the *final graph.*

Recall that process p_j does not suspect process p_i if it receives pairs labeled with p_i periodically. Roughly, what we show in Lemma 6 is that, once the topology of the graph stabilizes and all the messages from crashed processes are delivered, all processes at distance k from a process p_i, eventually receive the pair $(i, n - k)$. First, let us illustrate with an example why it is important to consider graph $G(t^\epsilon)$.

Consider the graph $G(t)$ with $t < t^\epsilon$ of Figure 4.1a with $n = 5$ and the hopbound values from process p_1 that every process has stored. Neighbors of p_1 receive pair $(1, 4)$, and the neighbors of neighbors of p_1 receive pair $(1, 3)$ and so on. In this graph, the distance from p_1 to p_4 is 2. Now, let us assume that only process p_2 fails. The final graph $G(t^\epsilon)$ is shown in Figure 4.1b. In $G(t^\epsilon)$, the distance from p_1 to p_4 is 3. As soon as all the messages from p_2 are delivered, it must happen that the value $hopbound_4[1] = 3$ expires, since p_4 will not receive again that pair. Then, eventually, every Δ units of time, p_4 will receive a message from p_5 including the pair $(1, 2)$, and that is the maximum $hopbound_1$ that p_4 can receive. This $hopbound_1$ in p_4 is not a coincidence, this value corresponds to $n - d(p_1, p_4) = 5 - 3 = 2$ with $d(p_1, p_4)$ the distance of p_1 to p_4 in $G(t^\epsilon)$. Lemma 6 shows formally this property.

A *correct path* from p_i to p_j is a path where the first process is p_i, the last process is p_j and all processes on it are correct. Lemma 6 shows that there is some time t such that $t > t^\epsilon$ after which correct processes connected by correct paths, satisfy the eventual



(a) Graph $G(t)$ with $t < t^\epsilon$ (b) Final graph $G(t^\epsilon)$

Figure 4.1: Graph G before and after failures.

strong accuracy property, i.e. eventually a correct process does not suspect a correct process in its same partition.

Lemma 6. *Let p_i and p_j be two correct processes in $V(G(t^\epsilon))$ such that $d(p_i, p_j) = m$ with $m < n$ in $G(t^\epsilon)$. Then, the following properties hold:*

1. *There is a time $t_q \geq t^\epsilon$ after which $hopbound_j[i]$ is equal to $n - m$ permanently ,*
2. *There is a time $t'_q \geq t^\epsilon$ after which p_j stops suspecting p_i permanently .*

Proof We prove this lemma by strong induction on the length of the path connecting p_i to p_j.

Base case: $m = 1$. For every correct processes p_i, p_j at distance 1 (that is, neighbors), the two properties are satisfied.

The proof of property 1 is as follows. When p_j receives an alive message from p_i, it goes through the pairs and stores only the hopbound values from the set of processes which are not direct neighbors, plus p_i (line 21). Since $n - 1$ is the greatest $hopbound_i$ value that p_j can receive, the first condition in line 22 is true and the $hopbound_j[i]$ is set to $n - 1$ (line 23). If another process p_j sends an alive message including a pair labeled with p_i, the condition in line 21 is not true, so line 23 is not reached, and the $hopbound_j[i]$ variable is not changed.

We now prove property 2. If p_j is wrongly suspecting p_i, it increases the timer in $timeout_j[i]$ (line 26) and eventually gets a value X such that $X > \Delta$ and stops changing. Let us consider the time before $timeout_j[i] = X$. If p_j is suspecting p_i, when p_j receives a message from p_i, the condition in line 24 is true, so p_j increases the $timeout_j[i]$ variable and sets the $suspect_j[i]$ variable to false. Since the $timeout_j[i]$ variable reached its upper bound and p_j receives a message from p_i in at most every Δ units of time, then condition in line 35 is never true, so the $suspect_j[i]$ variable does not change. Then, p_j stops suspecting p_i permanently.

Induction step: Assume that for all correct processes p_ℓ such that $d(i, \ell) <= m-1$ in $G(t^\epsilon)$, the properties are true. Let p_j be a correct process such that $d(i, j) = m$ in $G(t^\epsilon)$ and let $\pi = p_i, ..., p_\ell, p_j$ be a minimum length path of correct processes connecting p_i to p_j. We prove that the properties are true for p_j.

The proof of both properties is as follows. By property 2 and property 1 of the inductive hypothesis, for all $t > max(t_\ell, t'_\ell)$, p_j includes the pair $(i, n-m)$ in its bag when it sends an alive message, since $n - m > 0$ and $suspect_\ell[i] = false$ (condition in line 11 is true). By Lemma 3, that bag is delivered to p_j at time at most $max(t_\ell, t'_\ell) + \Delta$. Since π is a minimum length path connecting p_i to p_j, the largest $hopbound_i$ that p_j can receive is $n - m$.

In case of $hopbound_j[i] > n - m$, i.e. p_i and p_j were connected by a path shorter than π, eventually p_j suspects p_i (since this path does not exist anymore) and second condition in line 22 is true so line 23 is executed, i.e. $hopbound_j[i] = n - m$.

Since p_j receives the maximum $hopbound_i$ that it can receive in at most every Δ units of time, if p_j is wrongly suspecting p_i, it increases the $timeout_j[i]$ (line 26) and gets the value X which reaches the upper bound Δ.

Let us consider the time before $timeout_j[i] = X$. If p_j is suspecting p_i, condition in line 24 is true. When p_j receives a message from p_j, process p_j sets $hopbound_j[i] = n-m$, increases the $timeout_j[i]$ and sets the $suspect_j[i]$ to false. Since the $timeout_j[i]$ reached its upper bound and p_j receives messages from p_j including the pair $(\ell, n - m)$ in at most $\Delta \leq X$ time, then condition in line 35 will never be true, so the $suspect_j[i]$ variable will not change, thus line 23 is only executed if first condition of line 22 is true, but the maximum $hopbound_i$ that can be received is $n - m$, so $hopbound_\ell[i] = n - m$ permanently. $\square_{Lemma\ 6}$

The following theorem proves one of the two requirements of an eventual perfect failure detector. The other requirement will be stated in Theorems 14 and 15.

Theorem 7 (Eventual Strong Accuracy)**.** *Let p_i and p_j be two correct processes. After t^f, if p_i and p_j are on the same partition, p_j eventually does not suspect p_i permanently.*

Proof Since p_i and p_j are on the same partition i.e. the same connected component, there exists a correct path of minimum length between p_i and p_j and vice versa in $G(t^f)$. Then, by property 2 of Lemma 6, there is a time after which p_j does not suspect p_i permanently. $\square_{Theorem\ 7}$

The following lemma, shows that the neighbors of a crashed process p_i satisfy strong completeness, i.e. neighbors of p_i eventually suspect p_i permanently.

Lemma 8. *Let p_i and p_j be two neighboring processes such that p_j is correct and p_i is faulty. There exists a time t after which p_j suspects p_i.*

Proof Before p_i fails, it sent a finite number of messages to p_j and p_j only changes $timeout_j[i]$ when an alive message from p_i arrives, so eventually $timeout_j[i]$ stops changing. Let t^ℓ the time when p_j receives the last message from p_i. By time $t = t^\ell + timeout_j[i]$ the timeout for a message from p_i expires and p_j sets $suspect_j[i]$ to true (line 35). This is permanent, since p_j only changes $suspect_j[i]$ if it receives a message from p_i, but it will not receive any other messages from p_i. $\square_{Lemma\ 8}$

Now we have to prove that after t^f, all correct processes suspect all crashed processes. In order to prove that, first we show that once a process p_i fails, the pairs labeled with

p_i eventually stop being sent by any correct process. Then, we show that there is a time $t > t^f$ after which all processes in $V(G(t))$ suspect all crashed processes. First, let us observe the following.

Observation 9. *By Lemma 6, we can conclude that the largest hopbound$_i$ that a process at distance k of p_i in G can receive is $n - k$.*

A process p_j suspects p_i if p_j does not receive a pair labeled with p_i before the timer expires. When p_i fails, processes know this information because as the time goes by, the hopbounds from p_i fade away in the network until the point where no process receives pairs labeled with p_i. Lemma 11 states this idea formally. We give an example of how this happens.

Let us consider graph G in Figure 5.1, with the hopbound value from p_1 that every process has stored.

When p_1 fails, as soon as the timer expires in p_3 and p_2, they suspect p_1. We draw in red processes that suspects p_1 in Figure 4.2b.

Then, when neighbors of p_1 suspect it, they do not include in their bags permanently pairs labeled with p_1. Thus, the *hopbound*$_1$ stored in p_4, p_5 and p_6 expires, and when one of these processes receives a pair labeled with p_1, it takes this new value (second condition in line 22). A possible execution is illustrated in Figure 4.2c. At time t^2, the longest *hopbound*$_1$ that a process could have stored is $n - 2 = 7 - 2 = 5$.

After pair $(1, 5)$ expires, the largest *hopbound*$_1$ that a process can have stored is $n - 3$, as illustrated in Figure 4.2d.

Thus, in the graph of Figure 4.2e, processes p_5, p_6 and p_7 keep a *hopbound*$_1 = 1$, which means that they do not include in their bags pairs labeled with p_1. Therefore, p_4 suspects p_1, so even if its *hopbound*$_1$ is greater than 1 it does not include a pair labeled with p_1. Then, by time t^* no process sends pairs labeled with p_1.

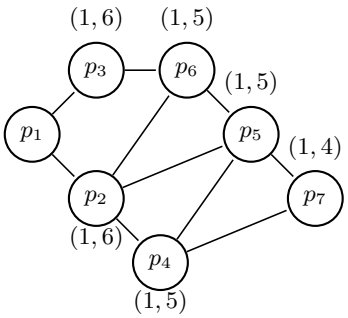
The following lemma proves that the hopbound of a crashed process eventually disappears.

Lemma 10. *After a process p_i crashes, there is a time t^1 such that for all $t > t^1$ correct neighbors of p_i do not include in their bags pairs labeled with p_i.*

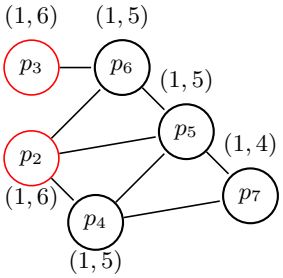
Proof By Lemma 6, only processes that are neighbors of p_i can have *hopbound*$_i = n-1$. Let t^d be the time where the last message from p_i arrives to a correct neighboring process p_j. By Lemma 8, there is a time $t^1 \geq t^d$ when all the correct neighbors of p_i suspect it. Therefore, for every $t \geq t^1$, neighbors of p_i do not include any pair labeled with p_i in their bags (line 11). $\square_{Lemma\ 10}$

Observe that neighbors are the only processes that can include the pair $(i, n-2)$ in their bags, because they have set *hopbound*$_i = n - 1$ permanently.

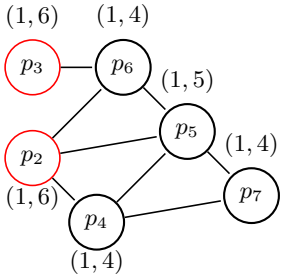
Lemma 11. *After a process p_i fails, for every $1 < k \leq n - 1$, there is a time t^k after which every process having hopbound$_i = n - k$ does not include in its bag the pair $(i, n-(k+1))$ and the maximum hopbound$_i$ that a correct process can have is $n-(k+1)$.*



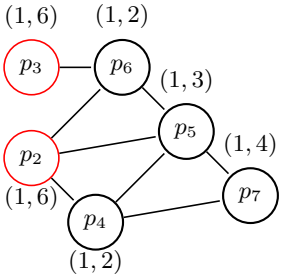
(a) Graph G at time $t > t^f$



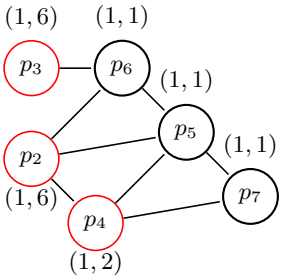
(b) Graph G at time t^1



(c) Graph G at time t^2



(d) Graph G at time t^3



(e) Graph G at time t^*

Figure 4.2: Spread of failure information of p_1.

Proof We prove the lemma by induction over k.

Base case: $k = 2$. Let us assume that there is a set $\mathcal{S}^2$ of correct processes such that $hopbound_i = n - 2$. By Lemma 10, neighbors of p_i are the only processes that can send $hopbound_i = n - 2$, and after time t^1 (given by Lemma 10) they do not send pairs labeled with p_i. Then, there is a time t^2 after which the timer for $n - 2$ in all the processes of $\mathcal{S}^2$ expires. Then, when a pair (i, m) arrives to some process in $\mathcal{S}^2$, m can be at most $n - 3$, so the maximum $hopbound_i$ that a correct process can have stored is $n - 3$.

Induction step: Assume that for $1 < k < n - 1$, after time t^k, processes having $hopbound_i = n-k$ do not include in their bags the pair $(i, n-(k+1))$ and the maximum $hopbound_i$ that a correct process can have stored is $n - (k + 1)$. Let $\mathcal{S}^{k+1}$ be the set of correct processes having $hopbound_i = n - (k + 1)$ and let $q \in \mathcal{S}^{k+1}$. Recall that process p_j includes a pair labeled with p_i if p_j does not suspect p_i and $hopbound_j[i] - 1 > 0$ (condition in line 11). In case that p_j suspects p_i, this Lemma is true because the first condition in line 11 is not true. Thus we have to prove the case where p_j does not suspect p_i. There are two cases.

Case 1: $k + 1 \neq n - 1$. By the inductive hypothesis, there is a time t^k after which processes having $hopbound_i = n - k$ do not send the pair $(i, n - (k + 1))$. Then, the $hopbound_i = n-(k+1)$ expires in every process in $\mathcal{S}^{k+1}$. When an alive message arrives with a pair (i, m), only the second condition of line 22 can be true, namely, a smaller $hopbound_i$ is stored. This value is smaller than $n-(k+1)$ since by inductive hypothesis, the maximum $hopbound_i$ that a process can have stored is $n - (k + 1)$. Therefore, at time t^{k+1} processes in $\mathcal{S}^{k+1}$ do not include the pair $(i, n - (k + 2))$ and the maximum $hopbound_i$ that a process can have stored is $n - (k + 2)$.

Case 2: $k+1 = n-1$. It is important to note that if a process p_j has $hopbound_j[i] = 1$ at any time, the second condition in line 11 is not true, so p_j does not include in its bag a pair labeled with p_i. Therefore, after time t^{k+1}, correct processes having $hopbound_i = 1$ do not send a message to its neighbors with a pair labeled with p_i.

$\square_{Lemma\ 11}$

Lemma 12. *After some process p_i crashes, there is a time t^* after which every correct process in $V(G(t^*))$ does not send a pair labeled with p_i.*

Proof This lemma is direct consequence of Lemma 11.

$\square_{Lemma\ 12}$

Let $P_1^{t^k}, ..., P_\ell^{t^k}$ be the set of partitions of $G(t^k)$. Lemma 13 shows that given two processes p_i and p_j at different partitions, p_j eventually stops receiving pairs labeled with p_i.

Lemma 13. *Let p_i be a correct process such that at time t^k, $p \in P_i^{t^k}$, and let $P_j^{t^k} \in G(t^k)$ such that $i \neq j$. Then, there is a time t after which all correct processes in $P_j^{t^k}$ do not send a pair labeled with p_i.*

Proof Let $t^d < t^f$ be the time when there was the last path π of length m connecting p_i to p_j, and let $t^k > t^d$ be the time when a $p_k \in \pi$ crashes causing that p_i and p_j are

in different partitions. Let $P_j^{t^k}$ be the partition where p_j is, and $p_\ell \in P_j^{t^k}$ the process that has the largest $hopbound_i$ with $hopbound_{p_\ell}[i] = n - m$.

Following the same argument of Lemma 11, there will be a time when the timer for $n - m$ expires and the maximum $hopbound_i$ that a process in $P_j^{t^k}$ can have is at most $n - (m + 1)$. Therefore, we can conclude that there exists a time t when all processes in the partition of p_j do not send any pair labeled with p_i.

$\square_{Lemma\ 13}$

The following two theorems show the strong completeness property: eventually every correct process suspects permanently every faulty process or processes that are not in the same partition.

Theorem 14 (Strong Completeness)**.** *Let p_i be a faulty process. There is a time t after which all correct processes suspect p_i permanently.*

Proof Let us consider t^d the time when p_i fails. By Lemma 11, there is a time t^* after which all correct processes in $V(G(t^*))$ stops sending messages labeled with p_i.

Let $r \in V(G(t^*))$. Process p_j only changes $timeout_\ell[i]$ when a message labeled with p_i arrives, so eventually $timeout_\ell[i]$ will stop changing.

Let $t^\ell = max(timeout_\ell[i])$ with $\ell \in V(G(t^*))$. By time $t^* + t^\ell$, the timer for a message labeled with p_i expires and every process p_j sets $suspect_\ell[i]$ to true (line 35). This is permanent, since p_j only changes $suspect_\ell[i]$ if it receives a pair labeled with p_i, but it will not receive any other pair. $\square_{Theorem\ 14}$

Recall that t^f is the earliest time when all the failures have occurred. Thus, if p_i is a process in a partition $P_i^{t^f}$, then p_i is correct.

Theorem 15 (Strong Completeness)**.** *Let $P_i^{t^f}, P_j^{t^f}$ be two distinct partitions at time t^f. Then, there is a time $t \geq t^f$ at which every (correct) process $q \in P_j^{t^f}$ suspects every $p \in P_i^{t^f}$ permanently.*

Proof The proof is similar to the proof of Theorem 14. First we focus on the time when the messages of p_i fade away in the network. Then, the proof is the same for both cases. By Lemma 13, every $j \in P_j^{t^f}$ stops sending messages labeled with p_i at time t^{**}. Following the same argument of Theorem 14, there is a time after which all processes $\ell \in P_j^{t^f}$ set $suspect_\ell[i]$ to true permanently (line 35).

$\square_{Theorem\ 15}$

Theorem 16. *Algorithm 8 implements $\Diamond P$ in an arbitrary network using messages of $O(n\ log\ n)$ size.*

Proof Strong completeness is given by Theorem 14 and Theorem 15. Eventual strong accuracy is given by Theorem 7. Every time a process p_i sends an alive message, it sends the variable bag_i, which contains n integers bounded by n, the number of identifiers in the network, which can be represented by $log\ n$ bits. Thus, the size of the messages used in the algorithm are bounded by $O(n\ log\ n)$.

$\square_{Theorem\ 16}$

4.2 In networks with unknown membership

In the model proposed in Section 4.1 we assumed that every process p_i knows the names of all the processes in the network. This is because without this assumption, no failure detector of the original hierarchy can be implemented as shown in [33]. Recently, more general models have been proposed, like *dynamic networks*, and networks with unknown membership e.g. [26]. Such works had to adapt the completeness and accuracy properties, and proposed a new failure detector class called $\Diamond S^{\mathcal{M}}$.

The algorithm proposed in Section 4.1.2 can be extended to work in a network with unknown membership. For that, we consider the same timing model of Section 4.1 and the communication model of [26], i.e. processes communicate each other by sending and receiving messages via a packet network that keeps the bounds given by ADD channels.

In a dynamic network, processes do not know the names of all the participants in the network. They learn the names dynamically, as processes show up and send alive messages to their neighbors. Because of this, every process keeps a list of processes it *knows*, namely, processes from which it has received messages. To extend completeness and accuracy for dynamic networks, we define these properties formally as in [26].

4.2.1 $\Diamond P^{\mathcal{M}}$ specification

Given an execution α, a process p_i is said to be *known* if there is a process p_j and a time t such that for all $t' \geq t$, p_i is in the known list of p_j. We define $\mathsf{known}(\alpha, t)$ as the set of all processes that are known at time t. We define the following sets:

- $CRASHED(\alpha) = \bigcup_{t \in \mathbb{N}} \mathsf{crashed}(\alpha, t)$
- $CORRECT(\alpha) = \Pi - CRASHED(\alpha)$
- $KNOWN(\alpha) = \bigcup_{t \in \mathbb{N}} \mathsf{known}(\alpha, t)$

In our original model, the network is partitionable, so we have to give a new definition of completeness and accuracy for partitionable and with unknown membership networks. For this, let us define $K(p_j) = \{p_i |$ there exists a time t in which p_i knows $p_j\}$. $\Diamond P^{\mathcal{M}}$ for partitionable networks is formally defined as follows. For each execution α, there is a time t^a such that for every $t > t^a$ the failure detector satisfies:

- **Strong completeness**: For every process $p_j \in CRASHED$ or $p_j \in CORRECT$ that is not in the same partition of $p_i \in CORRECT \cap K(p_j)$ in $G(t)$, p_i suspects p_j at time t
- **Eventual strong accuracy**: For every process $p_j \in CORRECT$ in the same partition of p_i in $G(t)$, p_i does not suspect p_j at time t

Strong completeness means that if a process p_j has crashed or if it is correct, but is in a different partition than a process p_i that does know about p_j, eventually p_i suspects p_j. Eventual strong accuracy means that if a process p_j is correct and is in the same partition that p_i is, eventually p_i does not suspect p_j.

4.2.2 Description of the algorithm

Algorithm 8 can be modified to implement $\Diamond P^{\mathcal{M}}$, with the changes described next.

The first time p_i receives through channel m a message sent by a process p_j, p_j and m become synonyms from a neighbor addressing point of view. As every process p_i knows nothing about the network, the longest path known by p_i initially is of length 1 (there might be some reachable process). Recall that a pair (j, ttl) is added to the bag of p_i if $hopbound_i[j]-1 > 0$. Then, when **Algorithm 5** starts, process p_i broadcasts a HB message included in its bag, the pair $(i, localhopbound)$ (line 8) with $localhopbound_i = 2$. This is to guarantee that neighbors forward the information of p_i to their neighbors. And as p_i *discovers* new processes in the network, namely, receives pairs from processes that it does not know, the *localhopbound* grows.

Thus, instead of fixing the hopbound of p_i to $n-1$, it grows dynamically as processes are discovered. The data structures used for storing timeouts, clock values, etc. need to be dynamic as well. In Algorithm 5 we use the array notation for convenience, but in an implementation, a more efficient (dynamic) data structure will be used.

New processes may be discovered when an alive message from a neighbor arrives. Processes that have been discovered are stored in the local variable *known*. When p_i receives an alive message from a neighbor p_j containing a bag, p_i goes through the pairs (ℓ, m) in the bag and it checks if p_j is a known process. If that is the first time that p_i receives a message labeled with p_j, p_i discovers p_j, i.e. it adds to its set of known processes (line 29), and adds a new entry in every variable (line 30), initialized as described in Algorithm 8.

Algorithm 5 shows only the required modifications; it includes only the lines that need to be modified in every module, as well as the new variables that are needed.

4.2.3 Correctness proof

To prove the correctness of Algorithm 5, we need to show that the implementation satisfies the **strong completeness** and **eventual strong accuracy** for partitionable networks with unknown membership.

First, we have to prove that the $localTTL$ of every process p_i is bounded by $n + 1$.

Lemma 17. *For every correct process p_i, $hopbound_i[i] \leq n + 1$.*

Proof Let p_i be a correct process. At the beginning of the execution $hopbound_i[i] = 2$ and this value only is changed every time that p_i receives a pair from a process that it does not know (line 22). Since $|\Pi| = n$, the maximum number of distinct processes that p_i can know is $n - 1$, so the line 22 can only be executed $n - 1$ times. Therefore, $hopbound_i[i] \leq 2 + n - 1 = n + 1$

$\square_{Lemma\ 17}$

We only present a sketch of the proof for completeness and accuracy. The proof is very similar to the one given in Section 4.1.3. We have to show that given two correct processes p_i and p_j connected by a correct path in the final graph, p_j eventually knows p_i and p_j eventually does not suspect p_i just as Lemma 6.

Algorithm 5 Code for process p_i

Constants
1: T
Variables
2: $clock_i() = 1$
3: $neighbors \leftarrow \emptyset$
4: $known_i \leftarrow \emptyset$ ▷ Set containing processes known by p_i
5: $hopbound_i[i] \leftarrow 2$ ▷ hopbound value of p_i

6: **every** T **time units of** $\mathsf{clock}_i()$ **do**
7: **begin:**
8: $bag_i = \{(i, hopbound_i[i])\}$
9: **for** each $j \in known_i - \{i\}$ **do**
10: **if** $suspect_i[j] = false$ and $hopbound_i[j] > 1$ **then**
11: $bag_i \leftarrow bag_i \cup \{(i, hopbound_i[j] - 1)\}$
12: **end if**
13: **end for**
14: **broadcast**(bag_i, i)
15: **end**

16: **when** ALIVE(bag_i) **is received from** p_j **through channel** m
17: $neighbors_i \leftarrow neighbors_i \cup j$
18: **begin:**
19: **for** each $(\ell, m) \in bag$ such that $\ell \notin neighbors \setminus \{j\}$ **do**
20: **if** $\ell \notin known$ **then**
21: DISCOVER(ℓ)
22: $hopbound_i[i] \leftarrow hopbound_i[i] + 1$
23: **else**
24: Include code of Algorithm 8 from line 22 to 29
25: **end if**
26: **end for**
27: **end**

28: **function** DISCOVER(ℓ)
29: $known_i \leftarrow known_i \cup \{\ell\}$
30: Add a new entry in $lastAlive_i$, $timeout_i$ and $suspect_i$ and initialize
31: **end function**

32: **when** $timeout_i[j] = clock() - lastAlive_i[j]$
33: **begin:**
34: $suspect_i[j] \leftarrow true$
35: **end**

For proving that all the pairs of a crashed process p_i eventually fade out from the network, the proof is similar to that of Lemma 11, but assuming that it is only true for processes that know p_i. Similarly with the proof of Lemma 13, only processes in a different partition than p_i that know p_i stops sending messages labeled with p_i. For processes that do not know p_i, this is true.

Chapter 5

Ω implementations

As it is discussed in Chapter 1, it has been shown that it is easy to implement the leader failure detector Ω given an implementation of $\Diamond P$. Given the vector output of $\Diamond P$, output the smallest identity of a non-faulty process for Ω. But the implementation of $\Diamond P$ requires a system with stronger conditions [12].

In this chapter it is presented an implementation of Ω in a much weaker system than the one assumed in Chapter 4. The most important contribution of this algorithm is that it achieves a smaller message size complexity and that we provide a mechanism for detecting ill paths as it is explained later.

This chapter also presents an implementation of Ω in networks with unknown membership that eventually uses messages of size $O(log\ n)$.

5.1 In networks with $\Diamond$ADD channels

This section presents Algorithm 7 that implements Ω, assuming each process knows n, the number of processes.

5.1.1 Ω specification

In this section we define formally the specification of Ω. For the first implementation we consider that there is a time τ after which there is a directed spanning tree (i) that includes all the correct processes and only them, (ii) its root is the correct process with the smallest identity, and (iii) its channels satisfy the $\Diamond$ADD property. This behavioral assumption is called *Span-Tree*. Since the network remains connected despite failures, the specification of the failure detector is the same as the presented in Chapter 2.

Let us observe that the local variables $leader_i$ of all the processes always contain a process identity. Hence, the proof must only show that the variables $leader_i$ of all the correct processes eventually converge to the same process identity, which is the identity of one of them.

Definition 10. *The leader failure detector satisfies the following properties [44]:*

- Validity: *Each invocation of* Ω *returns a process name*

- Eventual leadership: $\forall t' \geq t$ *and for some correct process* p_i*, every invocation of* Ω *for every correct process returns* p_i

5.1.2 Description of the algorithm

This algorithm uses local variables at each process, and describes the statements each process has to execute. A parameter T denotes an arbitrary duration. Its value affects the efficiency of the algorithm, but not its correctness[1].

The algorithm uses a single type of message denoted ALIVE. Such a message carries two values: a process identity and an integer $x \in \{2, \ldots, n-1\}$. In the following, "*" stands for any process identity. A message ALIVE$(*, n-1)$ is called *generating* message, while a message ALIVE$(*, n-k)$ such that $1 \leq k < n-1$, is called *forwarding* message. Moreover, the value $n-k$ is called *hopbound value*.

When process p_i starts the algorithm, it proposes itself as candidate to be leader. It sends a generating ALIVE$(i, n-1)$ message to its neighbors every T time units.

When a process p_i receives an ALIVE$(j, n-k)$ message such that $1 \leq k < n-1$, it learns that (a) p_j is candidate to be leader, and (b) there is a path with k hops from p_j to itself. If $j < i$, p_i adopts p_j as current leader, and forwards messages ALIVE$(j, n-(k+1))$ to its neighbors. It follows that a generating message ALIVE$(j, n-1)$ (which can be issued only by p_j) can give rise to a finite number of forwarding messages ALIVE$(j, n-2)$, ALIVE$(j, n-3)$,..., possibly up to ALIVE$(j, 2)$.

The hopbound stands for "upper bound on the number of forwarding" that –due to the last message ALIVE$(j, -)$ received by p_i– the message ALIVE$(j, -)$ sent by p_i has to undergo to be received by all processes. It is similar to a *time-to-live* value.

As it is possible that there are several paths from p_j to p_i of different lengths, p_i can receive different hopbound values of forwarding messages ALIVE$(j, n-k)$ with leader j. As it does not know which of those paths will satisfy the $\Diamond$ADD property, p_i manages a timer for each value of $n-k$ for each potential leader. In this way, p_i can associate increasing penalties with each hopbound value, namely, every time a hopbound value does not arrive on time, its penalty is increased. Assuming p_j will be the elected leader, p_i selects a hopbound value associated with p_j with the smallest penalty, which allows p_i to identify a path satisfying the $\Diamond$ADD property from an ALIVE$(j, n-k)$ message.

Each process p_i manages the following local variables.

- $in\_neighbors_i$ (resp., $out\_neighbors_i$) is a (constant) set containing the identities of the processes p_j such that there is channel from p_j to p_i (resp., there is channel from p_i to p_j).
- $leader_i$ contains the identity of the elected leader.
- $timeout_i[1..n, 1..n]$ is a matrix of timeout values and $timer_i[1..n, 1..n]$ is a matrix of timers, such that the pair $\langle timer_i[j, n-k], timeout_i[j, n-k] \rangle$ is used by p_i to monitor the elementary paths from p_j to it whose length is k.

[1] If T is too big, the failure detection of a process currently considered as a leader can be delayed. On the contrary, a too small value of T can entail false suspicions of the current eventual leader p_j until the corresponding timer $timer_i[j]$ has been increased to an appropriate timeout value.

- $hopbound_i[1..n]$ is an array of non-negative integers; $hopbound_i[i]$ is initialized to n, while each other entry $hopbound_i[j]$ is initialized to 0. Then, when $j \neq i$, $hopbound_i[j] = n - k \neq 0$ means that, if p_j is currently considered as leader by p_i, the information carried by the last message ALIVE$(j, n-1)$ sent by p_j to its out-neighbors (which forwarded ALIVE$(j, n-2)$ to their out-neighbors, etc.) went through a path[2] of k different processes before being received by p_i. The code executed by p_i when it receives a message ALIVE$(j, -)$ is detailed in Section 5.1.3.

- $penalty_i[1..n, 1..n]$ is a matrix of integers such that p_i increases $penalty_i[j, n-k]$ each time the $timer_i[j, n-k]$ expires. It is a penalization counter monitored by p_i with respect to the elementary paths of length k starting at p_j and ending at p_i.

- $not\_expired_i$ is an auxiliary local variable.

5.1.3 Algorithm

As many other leader election algorithms, Algorithm 7 elects the process that has the smallest identity among the set of correct processes. It is made up of three main sections: the one that generates and forwards the ALIVE() messages, the one that receives ALIVE() messages and the one that handles the timer expiration. Every section is described in detail below.

Launching the algorithm: The processes are not required to launch the algorithm simultaneously. Actually this is impossible as, even if their local clocks progress to the same speed, they are not initially synchronized.

It is assumed that any number of processes x, $1 \leq x \leq n$, start independently the algorithm[3]. Other processes start the algorithm when receiving an ALIVE() message for the first time. When this occurs, a process executes its initialization part before processing the message.

Initialization (Lines 2-9): Initially, each process p_i is candidate to be the leader, which is locally encoded by $leader_i = i$ (line 2). Process p_i also assigns n to $hopbound_i[i]$, sets $timer_i[i, n]$ to $+\infty$ (so this timer can never expire).

Then (lines 4-8), for each $j \neq i$, p_i assigns arbitrary positive integer values to each $timeout_i[j, x]$ for $1 \leq x \leq n$, sets the entries $timer_i[j, x]$ to the associated timeout values, initializes the penalties to -1, and assigns 0 to each $hopbound_i[j]$. (While the algorithm is correct whatever the initial timeout values, values obtained from previous experiments can be used to obtain algorithms instances which allow to expedite eventual leader election.)

Generating and forwarding messages (Lines 9-16): Every T time units of its local clock $\mathsf{clock}_i()$, a process p_i sends the message ALIVE$(leader_i, hopbound_i[leader_i] - 1)$. As previously defined, this message is a generating message if $leader_i = i$ and a

[2] In the graph theory, such a cycle-free path is called an *elementary* path.

[3] As in many other algorithms, since early on e.g. [13].

Algorithm 7 Eventual leader election in the $\Diamond$ADD model with known membership

Constants

1: $out\_neighbors, T, n$

Variables

2: $clock_i() = 1$
3: $leader_i \leftarrow i$; $hopbound_i[i] \leftarrow n$; set $timer_i[i, n]$ to $+\infty$;
4: **for** $j \in \{1, \cdots, n\} \setminus \{i\}$ and each $x \in \{1, \cdots, n\}$ **do**
5: $timeout_i[j, x] \leftarrow$ any positive integer;
6: set $timer_i[j, x]$ to $timeout_i[j, hb]$;
7: set $penalty_i[j, x]$ to -1; $hopbound_i[j] \leftarrow 0$
8: **end for**

9: **every** T units of time of $clock()_i$ **do**
10: **begin:**
11: **if** $(hopbound_i[leader_i] > 1)$ **then**
12: **for** each $j \in out\_neighbors_i$ **do**
13: send ALIVE$(leader_i, hopbound_i[leader_i] - 1)$ to p_j
14: **end for**
15: **end if**
16: **end**

17: **when** ALIVE$(\ell, hb \leftarrow n - k)$ **such that** $\ell \neq i$ **is received** ▷ from a process in $in\_neighbors_i$
18: **begin:**
19: **if** $(\ell \leq leader_i)$ **then**
20: $leader_i \leftarrow \ell$;
21: **if** ($[timer_i[leader_i, hb]$ expired) **then**
22: increase the value of $timeout_i[leader_i, hb]$
23: **end if**
24: set $timer_i[leader_i, hb]$ to $timeout_i[leader_i, hb]$;
25: $not\_expired_i \leftarrow \{x \mid timer_i[leader_i, x]$ not expired $\}$
26: $hopbound_i[leader_i] \leftarrow \max\{x \in not\_expired$ with smallest non-negative $penalty_i[leader_i, x]\}$
27: **end if**
28: **end**

29: **when** $timer_i[leader_i, hb]$ **expires** and $(leader_i \neq i)$ **do**
30: **begin:**
31: $penalty_i[leader_i, hb] \leftarrow penalty_i[leader_i, hb] + 1$;
32: **if** $\big(\wedge_{1 \leq x \leq n} ([timer_i[leader_i, x]$ expired$)\big)$ **then**
33: $leader_i \leftarrow i$;
34: **else**
35: same as lines 25-26
36: **end if**
37: **end**

forwarding message if $leader_i \neq i$ (in this case it is the forwarding of the last message ALIVE$(leader_i, hopbound)$ previously received by p_i). This message sending is controlled by the predicate of line 11, namely, it occurs only if $hopbound_i[leader_i] > 1$. The message

sent is then ALIVE($leader_i, hopbound_i[leader_i] - 1$).

The message forwarding is motivated by the fact that, if $hopbound_i[leader_i] > 1$, maybe processes have not yet received a message ALIVE($leader_i, -$) whose sending was initiated by $leader_i$ and then forwarded along paths of processes (each process having decreased the carried hopbound value) has not reached all the processes. In this case, p_i must participate in the forwarding. To this end, it sends the message ALIVE($leader_i, hopbound_i[leader_i] - 1$) to each of its out-neighbors (line 12).

Let us observe that during the anarchy period during which, due to the values of the timeouts and the current asynchrony, channel behavior and process failure pattern, several generating messages ALIVE($*, n-1$) can be sent by distinct processes (which compete to become leader) and forwarded by the other processes with decreasing hopbound values. But, when there are no more process crashes and there are enough directed channels satisfying the ADD property, there is a finite time from which a single process (namely, the correct process p_ℓ with the smallest identity) sends messages ALIVE(ℓ, n) and no other process p_j sends the generating message ALIVE(j, n).

Message reception (Lines 17-28): When a process p_i such that $leader_i \neq i$ receives a (generating or forwarding) message ALIVE(ℓ, hb), it discards it if $\ell > leader_i$ (predicate of line 19). This is due to the fact that p_i currently considers $leader_i$ as leader, and the eventual leader must be the correct process with the smallest identity. If $\ell \leq leader_i$, p_i considers ℓ as its current leader (line 19). Hence, if $\ell < leader_i$, p_ℓ becomes its new leader, while its current leader does not change if $\ell = leader_i$.

Then, as the message ALIVE(ℓ, hb) indirectly comes from $leader_i = \ell$ (which generated ALIVE($\ell, n-1$)) through a path made up of $k = n - hb$ different processes, p_i increases the associated timeout value if the timer $timer_i[leader_i, hb]$ expired before it received the message ALIVE(ℓ, hb) (line 21). Moreover, whether $timer_i[leader_i, hb]$ expired or not, p_i resets $timer_i[leader_i, hb]$ (line 24) thereby starting a new monitoring session with respect to its current leader and the elementary paths of length hb from $leader_i$ to it.

The role of the timer $timer_i[\ell, hb]$ is to allow p_i to monitor p_ℓ with respect to the forwarding of the messages ALIVE(ℓ, hb) it receives such that $hb = n - k$ (i.e., with respect to the messages received from p_j along elementary paths the length of which is k).

Finally, p_i updates $hopbound_i[leader_i]$. To update $hopbound_i[leader_i]$, p_i first computes the value of $not\_expired_i$ (line 25) which is a bag of elementary path length x such that $timer_i[leader_i, n - x]$ is still running[4]. The idea then is to select the less penalized path (hence the "smallest non-negative value" at line 26). But, it is possible that there are different elementary paths of lengths x_1 and x_2 such that we have $penalty_i[leader_i, n - x_1] = penalty_i[leader_i, n - x_2]$. In this case, in a conservative way, $\mathsf{max}(n - x_1, n - x_2)$ is selected to update the local variable $hopbound_i[leader_i]$.

Timer expiration (Lines 29-37): Given a process p_i, when the timer currently monitoring its current leader through a path of length $k = n - hb$ expires (line 29), it

[4]A bag (also called multiset or pool) is a "set" in which the same element can appear several times. As an example, while $\{a, b, c\}$ and $\{a, b, c, b, b, c\}$ are the same set, they are different bags.

increases its $penalty_i[leader_i, n-k]$ entry (line 31).

The entry $penalty_i[j, n-k]$ is used by p_i to cope with the negative effects of the channels which are on elementary paths of length k from p_j to p_i and do not satisfy the ADD property. More precisely we have the following. If, while p_i considers p_j is its current leader (we have then $leader_i = j$), and $timer_i[j, n-k]$ expires, p_i increases $penalty_i[j, n-k]$. The values in the vector $penalty_i[j, 1..n]$ are then used at lines 25-26 (and line 35) to update $hopbound_i[leader_i]$ which (if p_j is the eventually elected leader) will contain the length of an elementary path from p_j to p_i made up of $\Diamond$ADD channels (i.e., a path on which $timer_i[j, n-k]$ will no longer expire).

Then, if for all the hopbound values, the timers currently monitoring the current leader have expired (line 32), p_i becomes candidate to be leader (line 33).

If one (or more) timer monitoring its current leader has not expired, p_i recomputes the path associated with the less penalized hopbound value in order to continue monitoring $leader_i$ (line 35).

5.1.4 Correctness proof

Lemma 18. *Let p_i and p_j be two correct processes connected by a $\Diamond$ADD channel, from p_i to p_j. There is a time after which any two consecutive messages received by p_j on this channel are separated by at most $\Delta = K \times T + D$ time units.*

Proof For the channel (p_i, p_j), let us consider a time from which it satisfies the ADD property. Due to the ADD property, the channel delivers then to p_j (at least) one message from every sequence of K consecutive messages sent by p_i. Moreover, this message takes at most D time units. This means that at most $(K-1)$ messages can be lost (or take more than D time units) between two messages from p_i delivered consecutively by p_j. As p_i sends a message every T clock ticks and the local clocks run at a constant speed, the maximal delay between the consecutive receptions by p_j of messages sent by p_i is $\Delta = K \times T + D$. $\square_{Lemma\ 18}$

Given any run r of Algorithm 7, let $\mathsf{correct}(r)$ denote the set of processes that are correct in this run and $\mathsf{crashed}(r)$ denote the set of processes that are faulty in this run.

Lemma 19. *Given a run r, there is a time t^a after which there are no messages* $\text{ALIVE}(i, n-a)$ *with $p_i \in \mathsf{crashed}(r)$ and $1 \leq a < n-1$.*

Proof The proof of this lemma is by induction over a.

Base case: $a = 1$. There is a time t^1 after which no process sends $\text{ALIVE}(i, n-1)$ messages with $p_i \in \mathsf{crashed}(r)$.

Let us remind that a generating message $\text{ALIVE}(i, n-1)$ can be sent only by process p_i. If p_i crashes, it sends a finite number of messages $\text{ALIVE}(i, n-1)$. As this is true for any process that crashes, there is a finite time t^1 after which generating messages are sent only by correct processes.

Induction step: Let us assume there is a time t^a after which no process sends messages $\text{ALIVE}(i, n-a)$ with $p_i \in \mathsf{crashed}(r)$ and $1 \leq a < n-1$. To show that there is

a time t^{a+1} after which no process sends ALIVE$(i, n-(a+1))$ with $p_i \in \mathsf{crashed}(r)$ and $1 < a+1 \leq n-1$, we consider two cases.

- Case 1: $a+1 \neq n-1$. Since channels neither create nor duplicate messages and processes send a finite number of forwarding messages ALIVE$(i, n-a)$ before t^a (as defined by the inductive assumption), there is a finite time at which every process p_j whose $timer_j[i, n-a]$ is running, is such that $timer_j[i, n-a]$ expires for the last time. When this occurs the predicate at line 32 is evaluated. If the predicate is true (i.e. all the timers for p_i expired), p_j proposes itself as leader. Hence, the lemma follows from the fact the next ALIVE message that p_j sends, cannot be ALIVE$(i, -)$.

  If the predicate at line 32 is not true, p_j computes a new hopbound value (with respect to p_i if p_j still considers it as its current leader), which is the largest hopbound value whose timer has not expired and which has the lowest penalty number (line 35). It follows from the inductive assumption that, after time t^a, no process sends ALIVE$(i, n-a)$ messages with $1 \leq a < n-1$. Then, the new hopbound value (with respect to p_i) must be at most $n-(a+1)$. So in the next ALIVE message, the largest hopbound value (with respect to p_i) that can be sent by any process p_j is $(n-(a+2))$, so no process p_j sends forwarding messages ALIVE$(i, n-(a+1))$.

- Case 2: $a+1 = n-1$. The proof of this case follows directly from the predicate at line 11 (namely $hopbound_j[i] > 1$), which prevents any process p_j to send a message ALIVE$(*, 1)$.

$\square_{Lemma\ 19}$

Theorem 20. *Given a run r satisfying the* Span-Tree *property, there is a finite time after which the variables $leader_i$ of all the correct processes contain the smallest identity $\ell \in \mathsf{correct}(r)$. Moreover, after p_ℓ has been elected, there is a finite time after which the only messages sent by processes are* ALIVE$(\ell, -)$ *messages.*

Proof Initially (as any other process) the correct process p_ℓ with the smallest identity considers itself leader (line 2). Then it can be demoted only at line 20 when it receives a message ALIVE$(j, -)$ such that $j < leader_\ell = \ell$ (line 19). As p_ℓ is the correct process with the smallest identity, it follows that such a message was sent by a faulty process p_j (that crashed after it sent the generating message ALIVE$(j, n-1)$). Due to Lemma 19, there is a finite time τ after which there are no more messages ALIVE$(j, -)$ such that $j < \ell$. Hence, whatever the faulty process p_j, there is time $\tau' > \tau$ at which all the timers $timer_i[j, hb]$ with $hb \leq n$, have expired, and then p_ℓ considers itself leader (line 33). Then, due to the predicate of line 19, it can no longer be locally demoted. Moreover, due to Span-Tree assumption, there is a path made up of correct processes connected by $\diamond$ADD channels from p_ℓ to any other correct process. Due to Lemma 18 it follows then that there is a finite time after which each correct process repeatedly receives messages ALIVE$(\ell, -)$ with some hopbound value. Due to lines 19-20, processes adopts p_ℓ as leader. Since processes are repeatedly receiving messages ALIVE$(\ell, -)$ with some

hopbound value, the predicate at line 32 cannot become true as at least one hopbound value is always arriving on time at every correct process.

After p_ℓ has been elected, any alive process p_i is such that forever $leader_i = \ell$ and $timer_i[\ell, -]$ for some hopbound value never expires. It follows that, at line 12, a process p_i can send ALIVE$(\ell, -)$, messages only. $\square_{Theorem\ 20}$

Theorem 21. *The size of a message is* $O(\log n)$.

Proof The proof follows directly from the fact that a message carries a process identity which belongs to the set $\{1, \cdots, n\}$ and a hopbound number *hopbound* such that $2 \leq hopbound \leq n-1$. $\square_{Theorem\ 21}$

5.1.5 Time complexity

Given a run r, let ℓ denote the smallest identity such that $\ell \in correct(r)$. Let t^r be the time after which:

1. All failures already happened
2. All $\Diamond$ADD channels satisfy their constants K and D

Lemma 22. *Let p_i be a correct process such that for every $t > t^r$, $hopbound_i[\ell] = n-k$. Then, for every correct process p_j such there is an $\Diamond$ ADD channel from p_i to p_j, $timeout_j[\ell, n-(k+1)] \leq \mathcal{C} + 2^{log(\lceil \Delta \rceil)}$ with $timeout_j[\ell, n-(k+1)] = \mathcal{C}$ before t^r.*

Proof Since p_i sends a message every T units of time, by Lemma 18, after t^r, the maximum delay between the consecutive reception of two messages from p_i to p_j is Δ. After t^r, the $timeout_j[\ell, n-(k+1)]$ stops changing when $\Delta \leq timeout_i[\ell, n-(k+1)]$, so it cannot happen again that p_j expires $timer_i[\ell, n-(k+1)]$. Therefore, the timeout is not incremented again. In the case in which $timeout_i[\ell, n-(k+1)] \geq \Delta$ before t^r, this lemma is true.

In the other case, consider that $\Delta \leq \lceil \Delta \rceil$ and then $2^{log(\Delta)} \leq 2^{log(\lceil \Delta \rceil)}$. Since the timeout increment under false suspicions is exponential, $timeout_i[\ell, n-(k+1)]$ needs to be incremented at most $\lceil log\ \Delta \rceil$ times for $\Delta \leq timeout_i[\ell, n-(k+1)]$. Once it is true that $\Delta \leq timeout_i[\ell, n-(k+1)]$, it cannot happen again that p_i expires $timer_i[\ell, n-(k+1)]$, so the timeout is not incremented again. Therefore $timeout_i[\ell, n-(k+1)] \leq \mathcal{C} + 2^{log(\lceil \Delta \rceil)}$.

$\square_{Lemma\ 22}$

Lemma 22 states that a timeout value is increased a finite number of times. Let t^c be the time after which all timeouts reached its maximum, namely, no timeout is increased again. The following claims refer to the communication graph at this time.

Lemma 23. *For every correct process p_i such that there is a minimum length path of $\Diamond$ ADD channels of length k from p_ℓ to p_i, $leader_i = \ell$ at time $t^c + (k \times \Delta)$.*

Proof Let $S^k = \{p_i |$ there is a minimum length path of $\Diamond$ ADD channels from p_ℓ to $p_i\}$ and let x be the maximum length of a minimum length path of $\Diamond$ ADD channels from p_ℓ to any correct process.

Base case: At time $t^c + \Delta$, every $p_i \in S^1$ has $leader_i = \ell$. Since after t^c no timer expires again, by Lemma 18, after Δ units of time, every $p_i \in S^1$ receives a message from p_ℓ including itself as the leader. Since p_ℓ is the correct process with the smallest identity, condition in line 19 is true and $leader_i = \ell$ after $t^c + \Delta$.

Induction step: Let us assume that for $1 \leq k < x$ and for every $p_i \in S^k$, $leader_i = \ell$ at time $t^c + (k \times \Delta)$. Let $p_j \in S^{k+1}$ be a neighbor of p_i, namely, p_j is connected to p_i by an $\Diamond$ ADD channel and $k+1$ is the length of the minimum length path of $\Diamond$ ADD channels connecting p_ℓ to p_j. There can be two cases for p_i.

1. $hopbound_i[\ell] = n - k$. In this case, since the maximum distance between any two processes is $n-1$, condition in line 11 must be true and in Δ units of time at most, p_j must receive an ALIVE$(\ell, n - (k+1))$ message from p_i. Since p_ℓ is the correct process with the smallest identity, condition in line 19 is true and $leader_j = \ell$.

2. $hopbound_i[\ell] = n - k'$ with $k' > k$. In this case, we have to show that $k' < (n-1)$ for the condition in line 11 to be true. The only way in which p_i can have $hopbound_i[\ell] = n - k'$ is because there is a path of length k' from p_ℓ to p_i. This path must be simple in order to $k' < (n-1)$, otherwise there must be a cycle in the path of length $n-1$ between p_ℓ and p_i. Let $p_\ell, q_1, q_2, ..., q_s, q_{s+1}, ..., q_c, q_s, ..., p_i$ be that path with a cycle. So it must be that at some time q_s has $hopbound_s[\ell] = n-s$ because it received ALIVE$(\ell, n-s)$ from q_{s-1} and then $hopbound_s[\ell] = n - (s+a)$ with a the length of the cycle that it received from q_c. But this cannot be forever since q_{s-1} keeps sending ALIVE$(\ell, n-s)$ to q_s, so eventually every $timer_m[\ell, n - (s-b)]$ with $s < m \leq a$ and $s+1 \leq b \leq c$ for every process in the cycle must expire, which produces that $n - (s+a)$ does not arrive again and eventually $penalty_s[\ell, n-s] < penalty_s[\ell, n-(s+a)]$. Then, in line 26, $hopbound_s[\ell] = n-s$. So, this case cannot happen, and therefore $k' < (n-1)$, condition in line 11 must be true and in Δ units of time at most, p_j must receive an ALIVE$(\ell, n - (k'+1))$ message from p_i.

In both cases, $leader_j = \ell$ after $t^c + ((k+1) \times \Delta)$. $\square_{Lemma\ 23}$

Theorem 24. *For every correct process p_i it takes $O(\mathcal{D} \cdot \Delta)$ time to have $leader_i = \ell$ with $\mathcal{D}$ the diameter of the graph.*

Proof The proof is direct from Lemma 23. $\square_{Theorem\ 24}$

5.1.6 Simulation experiments

This section presents simulation experiments related to the performance predicted by Theorem 24 of Algorithm 7. Only a few experiments are presented, a more detailed

experimental study is beyond the scope of this thesis. Our experiments show that a leader is elected in time proportional to the diameter of the network, in two network topologies: a ring and a random regular graph of degree 3.

Considering the constants K and D satisfied by an $\Diamond$ ADD once it stabilizes, Lemma 18 shows that for a given T (the frequency with which the messages are sent), then $\Delta = K \times T + D$ is an upper bound on the time of the consecutive reception of two messages by a process. According to Theorem 24, the time to elect a leader is proportional to the diameter of the network, where the K, D and T determine the slope of the function.

For the (time and memory) efficiency of the experiments we assume some simplifying assumptions, which seem sufficient for a preliminary illustration of the results:

- All the channels are $\Diamond$ ADD to avoid the need of a penalization array
- All the messages are delivered within time at most D or not delivered at all. This is sufficient to illustrate the convergence time to a leader. Additional experimental work is needed to determine the damage done by messages that are delivered very late
- We selected $K = 4$, $D = 12$ and $T = 1, 5, 10$

Convergence experiments

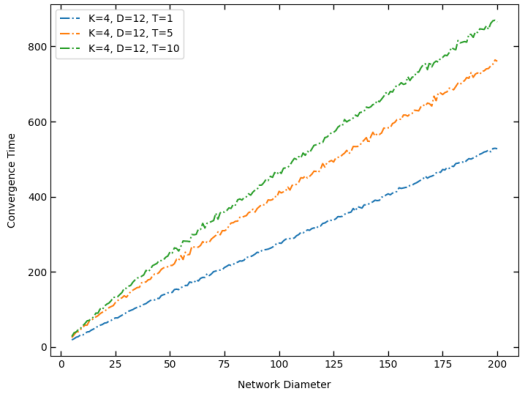
The experiments of the ring in Figure 5.1(a) and Figure 5.1(b), are when the probability of a message being lost is 1%, and 99% respectively. The case of a random graph of degree 3 up to 50,000 nodes is in Figure 5.1(c) when the probability of a message being lost is 1%. These experiments verify that indeed the convergence time is proportional to the diameter. The constants appear to be smaller than Δ, the one predicted by Theorem 24.

Simulation details

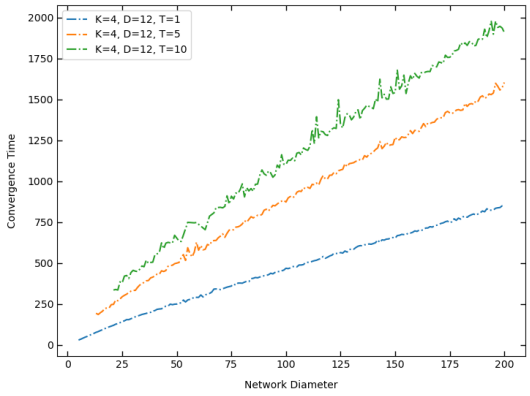
We performed our simulation results in a 48 core machine with 256GB of memory, using a program based on the Discrete Event Simulator *Simpy*, a framework for Python. We used the packet NetworkX to model graphs composed of ADD channels. To generate the ring and the random regular graph networks we used:

- *networkx.generators.classic.cycle_graph*
- *networkx.generators.random_graphs.random_regular_graph*

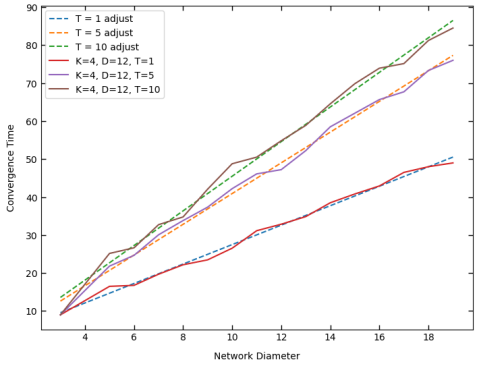
For the ring simulations, experiments were performed for each n from 10 up to 400 nodes, and taking the average of 10 executions, for each value of n. For the random regular networks, the degree selected was 3, and experiments starting with n starting in 100, up to $10,000$, taking the average of 5 executions. The n was incremented by 100 to reach 10,000 and from then on until 50,000 we incremented n by 10,000 each time. A performance impediment was indeed the large amount of memory used.



(a) A ring with drop rate of 1%



(b) A ring with drop rate of 99%



(c) A 3-regular random graph with drop rate of 1%

Figure 5.1: Convergence time in terms of the network diameter

The convergence time curves we obtained for the ring experiment are functions of the form $f(x) = c \cdot x$, where x represents the diameter of the network, and the constant c is, roughly, between 2.5 and 4.5 as T goes from 1 to 10. While for the random regular networks, we again got a constant that doubled in size, roughly, as T goes from 1 to 10. This behavior seems to be better than the one predicted by Theorem 24, which says that the constant c should have grown 10 times.

Re-election convergence simulation

If an elected leader fails, we would like to know in how much time a new leader is elected.

Note that the $\Diamond$ ADD channels can arbitrarily delay the delivery of some messages. This condition has a great impact in the time it takes to Algorithm 7 to change a failed leader. For the following simulations again we assume that all the messages are delivered within time at most D or not delivered at all. But note that in a realistic scenario, we can ease the impact of the arbitrarily delayed messages by adding a timestamp to every message and keeping track for every neighbor of this timestamp. If the timestamp of the recently received message is smaller than the current one, just ignore the message. This timestamp does not have a bound, but if we use an integer and increase it by one every second that a message is sent, this integer can hold on up for a century without overflowing [5]. By adding an integer to the message, we keep messages of size $O(log\ n)$.

For the simulation of Figure 5.2 we selected $K = 4$, $D = 12$, $T = 1$ and the probability of a message being lost is 1% . We performed this simulation on a ring. The algorithm starts at time t_0 and continues its execution till the average time in which a leader is elected (the curve represented in orange). In this time, the candidate to be the leader fails and then a timer from an external observer is started in every process. This timer is used to know the average time needed for each process to discard the failed leader (curve represented in purple) and then converge to a new leader (curve represented in blue). This experiment verify that indeed the convergence time after the current leader fails is proportional to the diameter since $\Delta = K \times T + D = 3 + 12 = 15$.

5.2 In networks with unknown membership

Here, while n exists and has a fixed value, it is no longer assumed that processes know it. Consequently, the processes have an "Unknown Membership" of how many and which are the processes in the network. Nevertheless, for convenience, the proposed algorithm still uses the array notation for storing the values of timers, timeouts, hopbounds, etc. (in an implementation dynamic data structures –e.g., lists– should be used).

Algorithm 9 solves eventual leader election in the $\Diamond$ ADD model with unknown membership, which means that, initially, a process knows nothing about the network, it knows only its input/output channels.

[5] An unsigned integer can be codified with 32 bits, so its maximum value can be 4294967296 $\simeq 4.3 \times 10^9$. A year has 31536000 $\simeq 3.1 \times 10^7$ seconds.

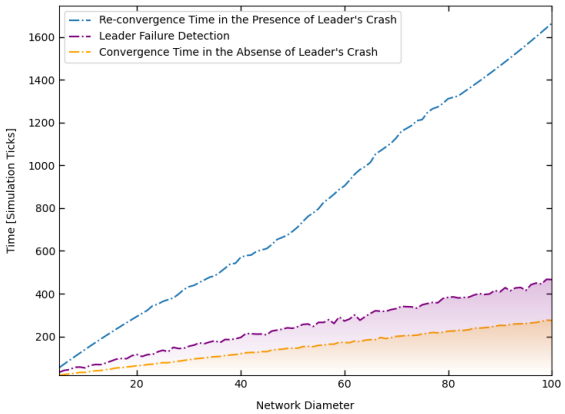


Figure 5.2: Convergence time for re-election

Our goal is to maintain the $O(\log\ n)$ bound on the size of the messages even in this model. It seems that it is not easy to come up with a minor modification of the first algorithm. For instance, a classic way of ensuring that forwarding the ALIVE message is cycle-free is to include the path information in the message along which the forwarding occurred, as done in the paper [34]. This would result in message sizes of exponential size, while assuming a slightly different model, we show how to eventually stay with $O(\log\ n)$ messages.

Furthermore, since we want the complexity to be $O(\log\ n)$ eventually, we need to design a mechanism that works as a *broadcast* in which once a process p_i knows a new process name from p_j, the later does not need to send to p_i the same information but only the leader information. The proposed mechanism in this paper is not the same as the proposed in [52] since we are preventing processes to send all the known names but eventually, only the leader information.

Since no process has knowledge about the number of participating processes, this number must be learned dynamically as the names of processes arrives. In order for the leader to reach every process in the network, there must be a path of $\Diamond$ ADD channels from every correct process to the leader. It follows that an algorithm for eventual leader election in networks with unknown membership cannot be a straightforward extension of the previous algorithm. More precisely, instead of the unidirectional channels and *Span-Tree* assumptions, Algorithm 9 assumes that (i) all the channels are bidirectional $\Diamond$ ADD channels, and (ii) the communication network restricted to the correct processes remains always connected (namely, there is always a path –including correct processes only– connecting any two correct processes).

In Algorithm 7, every process p_i uses n to initialize its local variable $hopbound_i[i]$ (which thereafter is never modified). In the unknown membership model, $hopbound_i[i]$ is used differently, namely it represents the number of processes known by p_i so far. So its initial value is 1. Then, using a technique presented in [52], $hopbound_i[i]$ is updated as processes know about each other: every time a process p_i discovers a new process identity it increases $hopbound_i[i]$.

5.2.1 Description of the algorithm

Initially each process p_i only knows itself and how many channels are connected to it. So the first thing p_i needs to do is communicate its identity to its neighbors. Once its neighbors know about it, p_i no longer sends its identity. The same is done with other names that p_i learns. For that, p_i keeps a *pending set* for every channel connected to it that tracks the information it needs to send to its neighbors. So initially, p_i adds the pair $(\texttt{new}, i)$ to every pending set.

During a finite amount of time, it is necessary to send an ALIVE() message to every neighbor without any constraint because the set of process names needs to be communicated to other processes. That is, information about a leader might be empty and the message only contains the corresponding pending set.

When process p_i receives an ALIVE() message from p_j, this message can contain information about the leader and the corresponding pending set that p_j saves for p_i. First, p_i processes the information contained in the pending set and then processes the information about the leader.

We next analyze how process p_i solves its subtasks.

How p_i learns new process names. If p_i finds a pair with a name labeled as new and is not aware of it, it stores the new name in the set $known_i$, increases its hopbound value, and adds to every pending set (except to the one belonging to p_j) this information labeled as new. In any case, p_i needs to communicate p_j that it already knows that information, so p_i adds this information to the pending set of p_j but labeled as an acknowledgment.

When p_j receives *name* labeled as an acknowledgment from p_i, i.e. $(\texttt{ack}, name)$, it stops sending the pair $(\texttt{new}, name)$ to it, so it deletes that pair from p_i's pending set. Eventually, p_i receives a pending set from p_j not including $(\texttt{new}, name)$, so p_i deletes $(\texttt{ack}, name)$ from p_j's pending set.

How p_i processes the leader information. As in Algorithm 7, every process keeps as leader a process with minimum id. Since it is assumed that all the channels are $\Diamond$ ADD, there is no need to keep a timer for every hopbound value or a penalty array. In this case, process p_i keeps the largest $n-k$, i.e. hopbound value that it receives from the process it considers to be the leader. If this value (or a greater one) does not arrive on time, p_i proposes itself as the leader. In case a smaller hopbound value of the leader arrives, it is only taken if its timer expired.

Each process p_i manages the following local variables.

- $leader_i$ contains the identity of the candidate leader.
- $hopbound_i[\cdot]$ is an array of natural numbers; $hopbound_i[i]$ is initialized to 1.
- $timeout_i[\cdot]$ and $timer_i[\cdot]$ have the meaning as in Algorithm 7. So, when p_i knows p_j, the pair $\langle timer_i[j], timeout_i[j]\rangle$ is used by p_i to monitor the sending of messages by p_j (which is not necessarily a neighbor of p_i).
- $known_i$ is a new set containing the processes currently known by p_i. At the beginning, p_i only knows itself.

- $out\_neighbors_i$ is a new set containing the local names of the channels connecting p_i to its neighbor processes. The first time p_i receives through channel m a message sent by a process p_j, p_j and m become synonyms from a neighbor addressing point of view.

- $pending_i[1, ..., k]$ is a new array in which, when p_i knows p_j, $pending_i[j]$ contains the pairs of the form $(label, id)$ that are pending to be sent through channel connecting p_i and p_j. There are two possible labels, denoted `new` and `ack`.

The code of Algorithm 9 addresses two complementary issues: the management of the initially unknown membership, and the leader election.

Initialization (Lines 2-7): Initially, each process p_i knows only itself and how many input/output channels it has. Moreover, it does not know the name of the processes connected to these channels (if any) and how many neighbors it has (the number of channels is higher or equal to the number of neighbors). So when the algorithm begins, it proposes itself as the leader and in the pending sets of every channel adds its pair $(\texttt{new}, i)$ for neighbors to know it.

Sending a message (Lines 8-17): Every T units of time, p_i sends a message through every channel m. In some cases the leader information is empty because of the condition of line 11. But in any case, it must send a message that includes information about the network that is included in the set $pending_i[j]$.

Receiving a message (Lines 18-49): When p_i receives a message (line 18) from process p_j (through channel m), at the beginning it knows from which channel it came and eventually knows from whom is from. When the message is received, the information included in *pending* (lines 21-37) is processed, and then the leader information is processed (lines 39-49).

Processing new information (Lines 21-37): The input parameter set *pending* includes pairs of the form $(label, id)$, where $label \in \{\texttt{new}, \texttt{ack}\}$ and id is the name of some process. When p_i processes the pairs that it received from p_j there can be two kinds of pairs. The first is a pair with label `new` (line 22), which means that p_j is sending new information (at least for p_j) to p_i. When this information is actually new for p_i (line 24) then, it stores this new name, increases its hopbound entry and adds to every pending set (but not the one from which it received the information) this new information (line 27).

In case that p_i already knows the information labeled as new for p_j (line 28), then p_i needs to check if it is included in the pending set to p_j this information as new too. If that is the case, then in deletes from $pending[m]$ this pair (line 29). In any case, p_i adds to the pending set the pair $(\texttt{ack}, k)$ for sending through the channel from where this message was received (line 30).

If p_i receives the pair $(\texttt{ack}, k)$ (line 33), then it deletes the pair $(\texttt{new}, k)$ from the set $pending_i[m]$, because the process that sent this pair, already knows k.

Processing the leader related information (Lines 39-49): If the leader related information is not empty, p_i processes it. As in the first algorithm, if the identity of

Algorithm 9 Eventual leader election in the $\Diamond$ ADD model with unknown membership

Constants

T, $out\_neighbors_i$

Variables

$clock_i() = 1$
$leader_i \leftarrow i$; $hopbound_i[i] \leftarrow 1$;
$known_i \leftarrow \{i\}$; $out\_neighbors_i$ initialized to the channels of p_i;
for each $m \in out\_neighbors_i$ **do**
    $pending_i[m] \leftarrow \{(\texttt{new}, i)\}$
end for

every T **time units of** $\mathsf{clock}_i()$ **do**
begin:
    for each channel $m \in out\_neighbors_i$ (let p_j be the associated neighbor) **do**
        if $(hopbound_i[leader_i] > 1)$ **then**
            send ALIVE$(leader_i, hopbound_i[leader_i] - 1, pending_i[j])$ to p_j
        else
            send ALIVE$(\perp, \perp, pending_i[j])$ to p_j
        end if
    end for
end

when ALIVE$(\ell, hb \leftarrow n - x, pending)$ **is received from** p_j **through channel** m
begin:
    $set_i \leftarrow \emptyset$;
    for each $(label, k) \in pending$ **do**
        if $(label = \texttt{new})$ **then**
            $set_i \leftarrow set \cup \{k\}$;
            if $(k \notin known_i)$ **then**
                $known_i \leftarrow known_i \cup \{k\}$; $hopbound_i[i] \leftarrow hopbound_i[i] + 1$;
                add an entry in $timeout_i$, $timer$, $hopbound_i$;
                add $(\texttt{new}, k)$ to every $pending[p]$ with $p \neq m$
            else if $((\texttt{new}, k) \in pending_i[m])$ **then**
                $pending_i[m] \leftarrow pending_i[m] \setminus (\texttt{new}, k)$ **end if**;
                $pending_i[m] \leftarrow pending_i[m] \cup (\texttt{ack}, k)$
            end if
        else
            $pending_i[m] \leftarrow pending_i[m] \setminus (\texttt{new}, k)$
        end if
    end for
    for each $(\texttt{ack}, k) \in pending_i[m]$ such that $k \notin set_i$ **do**
        $pending_i[m] \leftarrow pending_i[m] \setminus \{(\texttt{ack}, k)\}$
    end for
    if $(\ell \leq leader_i$ and $\ell \neq i)$ **then**
        $leader_i \leftarrow \ell$;
        if $(hb \geq hopbound_i[leader_i])$ $\vee$ $(timer_i[leader_i]$ expired$)$ **then**
            $hopbound_i[leader_i] \leftarrow hb$
            if $([timer_i[leader_i]$ expired$)$ **then**
                increase the value of $timeout_i[leader_i]$;
            end if
            set $timer_i[leader_i]$ to $timeout_i[leader_i]$
        end if
    end if
end

when $(timer_i[leader_i]$ expires$)$ **do** $leader_i \leftarrow i$

the proposed leader is smaller than the current one, then it is set as p_i's new leader (line 40). Then, it processes the hopbound. If the recently arrived hopbound is greater than the one currently stored, then the recently arrived is set as the new hopbound (line 42). If the timer for the expected leader expired, it needs more time to arrive to p_i, so when the timeout is increased (line 44) and the timer is set to timeout (line 46).

Deleting pairs (Lines 28, 33 and 36): If some process p_i wants to send some information k to p_j, it adds to the pending set of p_j the pair $(\texttt{new}, k)$. When p_j receives this pair, it looks if it is already in its set, in that case, it deletes the pair from p_i's pending set (line 28). Then, p_j adds an $(\texttt{ack}, k)$ to the pending set of p_i. As soon as p_i receives this pair from p_j, it deletes from p_j's pending set the pair $(\texttt{new}, k)$ (line 33). So the when p_j receives a pending set from p_i without the pair $(\texttt{new}, k)$, it means that p_i already received the acknowledgment message, so p_j deletes $(\texttt{ack}, k)$ from p_i's pending set (line 36).

Timer expiration (Line 50): When the timer for the expected leader expires, p_i proposes itself as the leader.

Notice that, when compared to Algorithm 7, Algorithm 9 does not use the local arrays $penalty_i[1..n, 1..n]$ employed to monitor the paths made of non-ADD channels.

5.2.2 Correctness proof

In the following we consider that there is a time τ after which no more failures occur, and the network is such that (i) there is a bidirectional path between every two correct processes, and (ii) its channels satisfy the $\Diamond$ ADD property. Assuming this, this section shows that Algorithm 9 eventually elects a leader despite initially unknown membership.

Lemma 25. *For any $p_i, p_j \in \mathsf{correct}(r)$ that are neighbors, eventually $known_i = known_j$.*

Proof Let p_i and p_j be two correct neighboring process. Assume that p_i communicates with p_j through $\Diamond$ADD channel a and on the other direction is channel b. At the initialization, p_i puts in all the *pending* sets the pair $(\texttt{new}, i)$ (line 6). Since it sends this set every T units of time through all the channels (lines 11-14), for Lemma 18, p_j eventually receives the pending set which contains at least the pair $(\texttt{new}, i)$. Then, for every pair $(label, k)$ that is received in p_j there are two cases: the pair contains new information or an acknowledgment.

When p_j receives a pair $(\texttt{new}, k)$ and it is the first time that it receives a pair with name k (condition in line 24), it adds k to its list of $known_j$ (line 25), adds to every pending set of every channel (except the one channel from which the message arrived from) the pair $(\texttt{new}, k)$ (line 27). In which case p_j already knows k, if p_j is trying to send this information as new to p_i, the pair is deleted from the pending set because p_i already knows this information (line 28). Finally, in either case p_j adds the pair $(\texttt{ack}, k)$ to the pending set of channel b, namely, the one from which p_j received the pair $(\texttt{new}, k)$ (line 30).

The second case is when the pair is an acknowledgment $(\texttt{ack}, k)$. Since acknowledgment messages are only received if previously p_j sent $(\texttt{new}, k)$ pair to p_i, then p_j deletes

the pair $(\texttt{new}, k)$ (if there is one) from the pending set of b (line 33). Thus no pair with label `new` is deleted until an acknowledgment is received or if it is received the same pair. But the acknowledgment is added only if the `new` pair was delivered to the receiver, meaning that p_j knows the same names that p_i.

Then, there cannot be some $k' \in known_i$ that p_j does not know eventually and vice versa. $\square_{Lemma\ 25}$

Lemma 26. *For every $p_i, p_j \in \mathsf{correct}(r)$, such that p_j is at distance d from p_i, there is a time t^d after which $i \in known_j$.*

Proof Let d' be the maximum distance between p_i and any other process. The proof of this lemma is by induction over d with $1 \leq d \leq d'$.

Base case: : $d = 1$. There is a time t^1 after which for every $j \in \mathsf{correct}(r)$ at distance 1 from p_i, eventually $i \in known_j$.

Since every process is connected by an $\Diamond$ ADD channel, eventually every neighbor p_j of p_i receives a message from it that contains in the pending set the pair $(\texttt{new}, i)$ since p_i added to every pending set that pair initially (line 8). Then, every p_j adds to its known set i, and adds to the pending set of p_i the pair $(\texttt{ack}, i)$. Process p_i does not delete pair $(\texttt{new}, i)$ from the pending set of p_j till it receives an $(\texttt{ack}, i)$ from p_j but this pair is only added if p_j received before the pair $(\texttt{new}, i)$ from p_i before. So eventually, every neighbor knows p_i.

Induction case: Let us assume there is a time t^d after which for every p_j at distance $d < d'$ from p_i, $i \in known_j$. We have to show that there is a time t^{d+1} after which for every p_j at distance $d + 1 \leq d'$ from p_i, $i \in known_j$

By the induction assumption, all processes p_j at distance d from p_i knows i. It means that before p_j knew p_i at some time p_j received the pair $(\texttt{new}, i)$ from some neighbor. Since p_j did not knew p_i, it added the pair $(\texttt{new}, i)$ to the pending set of every neighbor. Then, eventually that pending set is sent to every neighbor of p_j, so eventually p_i is known by processes at distance $d + 1$. $\square_{Lemma\ 26}$

Lemma 27. *Given a run r, there is a finite time t^a after which there are no messages* $\text{ALIVE}(i, k - a, pending)$ *with $p_i \in \mathsf{crashed}(r)$ such that $1 \leq a < k \leq n$.*

Proof First, note that the only process that can change the entry $hopbound_i[i]$ is p_i when it knows a new process (line 25). Since it only knew a finite number of processes before it failed, then the entry $hopbound_i[i]$ is finite and has an upper bound. Let us call k the value $hopbound_i[i]$ had before it failed.

The proof of this lemma is the same that for Lemma 19, by strong induction over a. Just note that when the timer expires is because ALIVE() message arrived with p_i as leader, no matter which hopbound of p_i is expected. $\square_{Lemma\ 27}$

What Lemma 25 shows is that eventually all the correct processes have the same set *known* and what Lemma 26 proves is that every correct process is in the *known* set of every correct process. Recall that for every process p_i, entry $hopbound_i[i]$ is increased

every time p_i knows a new process, namely, $hopbound_i[i]$ contains the cardinality of $known_i$. Then, we can conclude that there is a time t^f after which for every correct process p_i, there is a constant $k \leq n$ such that $hopbound_i[i] = k$, namely, each process has the same hopbound. Let τ be the time after which all the failures already happened. Let ℓ be the smallest identity in $\mathsf{correct}(r)$.

Lemma 28. *Let $p_i \in \mathsf{correct}(r)$ at distance d from p_ℓ after t^f with $0 \leq d \leq n-1$. There is a time $t > \tau$ and $t > t^f$ after which $hopbound_i[\ell] = k - d$ and $leader_i = \ell$ permanently.*

Proof We prove this lemma by strong induction on the length of the path connecting p_ℓ to p_i.

Base case: $d = 0$. For process p_ℓ, the two properties are satisfied.
Due to Lemmas 25 and 26, it is true that $hopbound_\ell[\ell] = k$. It can not be changed by any process since the only processes that can write $hopbound_\ell[\ell] = k$ is itself.

If $leader_\ell = \ell'$ such that $\ell' < \ell$, it means that $p_{\ell'}$ failed since it is assumed that ℓ is the smallest index of a correct process. Due to Lemma 27, there is a time after which p_ℓ stops receiving messages with leader $p_{\ell'}$. Eventually, the timer for p'_ℓ expires and p_ℓ proposes itself as the leader. This can only happen a finite number of times since the number of participating processes and the number of messages sent are finite.

Let us consider the last time in which the timer for receiving a message including a pair with $p_{\ell'}$ expires, then line 50 is executed, so $leader_\ell = \ell$. Since p_ℓ is the correct process with the minimum identity, it does not execute line 40 again.

Inductive case: Let us assume that there is a time after which $hopbound_j[\ell] = n-(m-1)$ and $leader_j = \ell$ permanently. Let p_i be a correct process at distance m from p_ℓ and let $\pi = p_\ell, ..., p_j, p_i$ be a minimum length path of correct processes connecting p_ℓ to p_i. We have to show that $hopbound_i[\ell] = k - m$ and $leader_i = \ell$.

By the induction assumption there is a time after which correct processes at distance $m-1$ satisfy the property. Since those processes send messages every T units of time, eventually process p_i receives a message including p_ℓ as the leader.

If $leader_i = p_{\ell'}$ such that $\ell' < \ell$ eventually $leader_i = p_\ell$ (same argument as base case).

Since p_i and p_j are connected by $\Diamond$ ADD channels, eventually p_i receives an ALIVE message from p_j including p_ℓ as the leader, so it sets $leader_i = \ell$ because condition in 40. Since there is a path of minimum length m connecting p_ℓ and p_i, $k-m$ is the largest hopbound value of p_ℓ that p_i can receive. Then, the first condition in line 41 is true and the $hopbound_i[\ell]$ is set to $k - m$.

If another process sends an ALIVE message including a pair $(p_\ell, m', pending)$ with $m < m'$ we have two cases.

- Case 1: The timer for p_ℓ is expired. In that case, second condition in line 41 is true, so $hopbound_i[\ell] = m$ (line 42) and the timeout is increased (line 44). Since processes are connected by $\Diamond$ ADD channels, there is a time after which messages from p_j arrive after at most Δ time to p_i, meaning that if the timer is expired,

every time that p_i receives an ALIVE message from p_j it increases the timeout. Eventually, the timeout gets a value $X > \Delta$ and stops changing.

This means that eventually this case is no longer true, namely the $timer_i[\ell]$ does not expire and as consequence $hopbound_i[\ell] = k - m$ and $leader_i = \ell$ does not change.

- *Case 2:* The timer for p_ℓ is not expired, so only first condition in line 43 can be true, since by the induction assumption, p_j sends $(p_\ell, n - m, pending)$ to p_i every T units of time, so the $hopbound_i[\ell] = k - m$ does not change and since we assumed that ℓ is the minimum index number, it must be that $leader_i = \ell$.

$\square_{Lemma\ 28}$

Lemma 29. *There is a time after which, at each correct process p_i and any of its channels m, the set $pending_i[m]$ becomes and forever remains empty.*

Proof Let p_i and p_j be two correct neighboring processes such that p_i is connected to p_j through channel a and in the other direction through channel b. We want to show that eventually $pending_i[a]$ ($pending_i[b]$) is empty.

Assume, without loss of generality, that p_i adds pair $(\texttt{new}, k)$ to $pending_i[a]$. Eventually p_j receives the pair $(\texttt{new}, k)$ from p_i. If this pair is already in $pending_j[b]$, then it is deleted (line 28) because p_i already knows k. Then p_j adds to $pending_j[b]$ pair $(\texttt{ack}, k)$ (line 30).

Eventually, p_i receives the pair $(\texttt{ack}, k)$ from p_j, so it deletes its $(\texttt{new}, k)$ pair from $pending_i[a]$ (line 33). Then, eventually p_j receives a message from p_i without the pair $(\texttt{new}, k)$, so p_j deletes the $(\texttt{ack}, k)$ from $pending_j[b]$ (line 36).

Process p_i can only add a finite number of `new` pairs in $pending_i[a]$, since the number of different processes is finite. Then, p_j can only add an `ack` pair to $pending_j[b]$ if it receives a `new` pair from p_i, namely, the number of `ack` pairs that it can add to $pending_j[b]$ is finite too.

All the `new` pairs are deleted as soon as the acknowledgment arrives, and the `ack` pairs are deleted as soon as the `new` pairs stops arriving. So eventually, every pending set of correct processes is empty. $\square_{Lemma\ 29}$

Theorem 30. *Given a run r, there is a finite time after which the variables $leader_i$ of all the correct processes contain forever the smallest identity $\ell \in \mathsf{correct}(r)$.*

Proof The proof follows from Lemma 28. $\square_{Theorem\ 30}$

Theorem 31. *Eventually, the size of a message is $O(\log n)$.*

Proof By Lemma 29, eventually each variable $pending_i[m]$ of a correct process is forever empty. So eventually, any ALIVE message carries a process identity which belongs to the set $\{1, \cdots, n\}$ and a hopbound number *hopbound* such that $2 \leq hopbound \leq n - 1$.

$\square_{Theorem\ 31}$

Chapter 6

Conclusions

The $\Diamond$ADD model has been studied in the past as a realistic, particularly weak communication model. A channel from a process p_i to a process p_j satisfies the ADD property if there are two integers K and D (which are unknown to the processes) and a finite time τ (also unknown to the processes) such that, after τ, in any sequence of K consecutive messages sent by p_i to p_j at least one message is delivered by p_j at most D time units after it has been sent.

In this thesis, we presented an algorithm to implement an eventually perfect failure detector in an arbitrary network connected by ADD channels using messages of $O(n \ \log \ n)$ size. For achieving small messages we used the time-to-live networking technique. In our basic implementation, the number and the name of processes in the network are known. Then we show how to extend the algorithm to work when each process knows initially only its neighbors. A process dynamically adapts its timeouts and failure information as it learns of known processes in the network.

Then, we assume a weaker model in which the ADD property is satisfied after some finite, but unknown, time and we proved that we can implement the leader failure detector on this model by providing an algorithm for implementing it. Again, we used the time-to-live values for achieving messages of size $O(log\ n)$. For this implementation is given a study of the performance and it showed that the more often the heartbeats are sent, the faster processes converge to the same leader, proving that having messages of small size is a great advantage.

Finally, we provide an algorithm for the eventual leader election for networks with unknown membership, which is something more than just an extension of the first algorithm since the assumed model is different. For the unknown membership model and the Ω implementation, we conjecture that it is necessary that the process identities are repeatedly communicated to the potential leader.

To the best of our knowledge, this is the first time that the technique of time-to-live values (well-known in the networking literature) is used for failure detectors. We make a first step in showing that it is a useful technique that leads to flexible failure detector implementations, of small message size.

In future work, we plan to study how to adapt it to more dynamic network scenar-

ios [26]. Specially we are interested in showing that our algorithms can be adapted to other partially synchronous models such as the one of [32], and even ones where channels are unidirectional or they may fail in one direction. Namely, in situations where reliable link protocol implementations are impossible, see e.g. [42].

Another interesting opportunity is the study of *communication-efficient* algorithms, meaning that after some finite time, only some linear number of edges carry out messages from the leader [4]. It would be interesting to investigate if there is a communication-efficient implementation for $\Diamond P$ or Ω in the ADD model for networks of arbitrary topology in which after some finite time, only a spanning tree carries messages from the leader being the root since all the current works are for complete networks and Ω implementations.

There are many interesting opportunities to explore tuning and extensions of our time-to-live approach. For example, for the concrete case of partitionable networks, while a node remains alive, everyone in its connected component keeps on receiving heartbeats, albeit of low intensity if it is a distant node, and hence the intensities can be used to estimate distances. It seems that in some situations a node does not need to send its current vector of TTL values to all its neighbors. To reduce load, it could send it in a round-by-round way, at the cost of increasing the time to detect failures. It is also of interest to investigate ways of tolerating very slow, old messages that can be delivered by an ADD channel, long after a process is dead. The algorithm works correctly, but these messages can affect performance by temporarily stopping to suspect a process that is long ago dead.

An interesting point for further research is the condition of only taking information of a neighbor from its ALIVEmessage for the $\Diamond P$ implementation. If this condition is removed, then we are giving priority to the underlying graph of time. This means that a process p_j possibly receives information of a neighboring process p_i from another neighbor p_k, which means that maybe is faster to know about p_i in p_j from another path that goes through p_k. If this condition is removed from our algorithm, it does not work, particularly 6 is not true but maybe it can be modified to work without this condition. A mechanism of penalization similar to the one presented in Algorithm 7 maybe can be used.

A much more detailed analysis would be needed for dynamic networks, but it seems the time-to-live approach could be adapted for this case too. An interesting open question is if the approach would be useful in dynamic networks with unknown membership, of the *time-free* type; such a failure detector (that does not rely on timers to detect failures) has been proposed in [27].

The performance analysis for Algorithm 3 has been proved experimentally in cycles and regular graphs. This experimental analysis is out of the scope of this work but interesting things emerged while doing it. For example, the amount of local memory needed for networks using hundreds of processes started to be a problem since matrices with a quadratic number of entries are used. It can be interesting to perform optimizations for memory use.

Finally, as it has been said in earlier chapters, failure detectors were proposed for circumvent the FLP impossibility result. Future work is to use the failure detectors pre-

sented in this thesis for solving the consensus using the eventual ADD model. It would be an interesting study because it does not seem easy to propose an implementation that copes with the weak properties that the ADD channels provide, in particular, the messages delivered with bounded, but unknown and arbitrarily large delay and because solving consensus in arbitrary graphs using failure detectors is a non explored work.

Bibliography

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *International Workshop on Distributed Algorithms*, pages 126–140. Springer, 1997.

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, 1999.

[3] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, number 2180 in Lecture Notes in Computer Science book series, pages 108–122, London, UK, UK, 2001. Springer-Verlag.

[4] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Comput.*, 21(4):285–314, 2008.

[5] Antonio Fernández Anta and Michel Raynal. From an asynchronous intermittent rotating star to an eventual leader. *IEEE Trans. Parallel Distrib. Syst.*, 21(9):1290–1303, 2010.

[6] Itziar Arrieta-Salinas, Federico Fariña, José Ramón González de Mendívil, and Michel Raynal. Leader election: From higham-przytycka's algorithm to a gracefully degrading algorithm. In *Sixth International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2012, Palermo, Italy, July 4-6, 2012*, pages 225–232, 2012.

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[8] Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

[9] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM*

SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983, pages 27–30, 1983.

[10] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[11] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[12] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[13] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[14] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.

[15] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on computers*, 51(5):561–580, 2002.

[16] Francis C. Chu. Reducing omega to diamond *W. Inf. Process. Lett.*, 67(6):289–293, 1998.

[17] Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Stabilizing leader election in partial synchronous systems with crash failures. *J. Parallel Distributed Comput.*, 70(1):45–58, 2010.

[18] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. A realistic look at failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 345–353. IEEE, 2002.

[19] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical report, 2003.

[20] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *23rd ACM Symposium on Principles of Distributed Computing*, number CONF, 2004.

[21] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[23] Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, and Michel Raynal. A distributed leader election algorithm in crash-recovery and omissive systems. *Inf. Process. Lett.*, 118:100–104, 2017.

[24] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[25] Felix C Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Computing Surveys (CSUR)*, 43(2):9, 2011.

[26] Fabíola Greve, Luciana Arantes, and Pierre Sens. What model and what conditions to implement unreliable failure detectors in dynamic networks? In *Proceedings of the 3rd International Workshop on Theoretical Aspects of Dynamic Distributed Systems*, pages 13–17. ACM, 2011.

[27] Fabíola Greve, Pierre Sens, Luciana Arantes, and Véronique Simon. Eventually strong failure detector with unknown membership. *Comput. J.*, 55(12):1507–1524, December 2012.

[28] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[29] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[30] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[31] Lisa Higham and Teresa M. Przytycka. A simple, efficient algorithm for maximum finding on rings. *Inf. Process. Lett.*, 58(6):319–324, 1996.

[32] Martin Hutle. An efficient failure detector for sparsely connected networks. In *Parallel and Distributed Computing and Networks*, pages 369–374, 2004.

[33] Ernesto Jiménez, Sergio Arévalo, and Antonio Fernández. Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2):60–63, 2006.

[34] Saptaparni Kumar and Jennifer L. Welch. Implementing $\Diamond P$ with bounded messages on a network of *ADD* channels. *Parallel Processing Letters*, 29(1):1950002:1–1950002:12, 2019.

[35] James F Kurose and Keith W Ross. *Computer networking: a top-down approach: international edition*. Pearson Higher Ed, 2013.

[36] Mikel Larrea, Antonio Fernández Anta, and Sergio Arévalo. Implementing the weakest failure detector for solving the consensus problem. *International Journal of Parallel, Emergent and Distributed Systems*, 28(6):537–555, 2013.

[37] Mikel Larrea, Sergio Arévalo, and Antonio Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In Prasad Jayanti, editor, *Distributed Computing, 13th International Symposium, Bratislava,*

Slovak Republic, September 27-29, 1999, Proceedings, volume 1693 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 1999.

[38] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *19th IEEE Symposium on Reliable Distributed Systems, SRDS'00, Nürnberg, Germany, October 16-18, 2000, Proceedings*, pages 52–59, 2000.

[39] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Trans. Computers*, 53(7):815–828, 2004.

[40] Mikel Larrea, Alberto Lafuente, Iratxe Soraluze, Roberto Cortiñas, and Joachim Wieland. On the implementation of communication-optimal failure detectors. In Andrea Bondavalli, Francisco Brasileiro, and Sergio Rajsbaum, editors, *Proc. Latin-American Symposium on Dependable Computing (LADC)*, number 4746 in Lecture Notes in Computer Science, pages 25–37, Berlin, Heidelberg, 2007. Springer.

[41] Mikel Larrea, Michel Raynal, Iratxe Soraluze Arriola, and Roberto Cortiñas. Specifying and implementing an eventual leader service for dynamic systems. *IJWGS*, 8(3):204–224, 2012.

[42] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[43] Achour Mostéfaoui and Michel Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *International Symposium on Distributed Computing*, pages 49–63. Springer, 1999.

[44] Michel Raynal. A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News*, 36(1):53–70, 2005.

[45] Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[46] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[47] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.

[48] Srikanth Sastry and Scott M. Pike. Eventually perfect failure detectors using ADD channels. In *Proc. 5th Int. Symposium on Parallel and Distributed Processing and Applications (ISPA)*, number 4742 in Lecture Notes in Computer Science, pages 483–496, Berlin, Heidelberg, 2007. Springer.

[49] Srikanth Sastry, Scott M. Pike, and Jennifer L. Welch. Crash-quiescent failure detection. In *Proceedings of the 23rd International Conference on Distributed Computing (DISC)*, volume 5805 of *Lecture Notes in Computer Science*, pages 326–340, Berlin, Heidelberg, 2009. Springer-Verlag.

[50] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981.

[51] Andrew S. Tanenbaum and David Wetherall. *Computer networks, 5th Edition*. Pearson, 2011.

[52] Karla Vargas, Sergio Rajsbaum, and Michel Raynal. An eventually perfect failure detector for networks of arbitrary topology connected with ADD channels using time-to-live values. *Parallel Process. Lett.*, 30(2):2050006:1–2050006:23, 2020.