

# Solving k-Set Agreement Using Failure Detectors in Unknown Dynamic Networks

Denis Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

► **To cite this version:**

Denis Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens. Solving k-Set Agreement Using Failure Detectors in Unknown Dynamic Networks. IEEE Transactions on Parallel and Distributed Systems, Institute of Electrical and Electronics Engineers, 2016. <hal-01354864>

**HAL Id: hal-01354864**

**<https://hal.inria.fr/hal-01354864>**

Submitted on 19 Aug 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Solving $k$ -Set Agreement Using Failure Detectors in Unknown Dynamic Networks

Denis Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

**Abstract**—The failure detector abstraction has been used to solve agreement problems in asynchronous systems prone to crash failures, but so far it has mostly been used in static and complete networks. This paper aims to adapt existing failure detectors in order to solve agreement problems in unknown, dynamic systems. We are specifically interested in the  $k$ -set agreement problem. The problem of  $k$ -set agreement is a generalization of consensus where processes can decide up to  $k$  different values. Although some solutions to this problem have been proposed in dynamic networks, they rely on communication synchrony or make strong assumptions on the number of process failures.

In this paper we consider unknown dynamic systems modeled using the formalism of Time-Varying Graphs, and extend the definition of the existing  $\Pi\Sigma_{x,y}$  failure detector to obtain the  $\Pi\Sigma_{\perp,x,y}$  failure detector, which is sufficient to solve  $k$ -set agreement in our model. We then provide an implementation of this new failure detector using connectivity and message pattern assumptions. Finally, we present an algorithm using  $\Pi\Sigma_{\perp,x,y}$  to solve  $k$ -set agreement.

**Index Terms**—Distributed systems, Dynamic networks, Failure detectors,  $k$ -Set agreement

## 1 INTRODUCTION

**D**YNAMIC distributed systems such as wireless or peer-to-peer networks pose new challenges to the field of distributed computing. In these systems, processes can join or leave the system during the run, and the communication graph evolves over time.

In unknown networks, processes are lacking initial information on system membership. Dynamic networks are often unknown, since it is difficult to know ahead of time which processes may join the system in the future.

Most of the existing distributed algorithms in the literature were meant for static, known networks and make assumptions that are unrealistic in the context of unknown dynamic networks: communication graphs are often assumed to be fully connected or even complete, and processes are expected to have full knowledge of the system membership. As a result, adapting existing protocols to unknown and/or dynamic networks is not trivial.

Agreement problems, and notably consensus, have been a lot less studied in dynamic networks than in static networks. In this paper we are particularly interested in the  $k$ -set agreement problem, which is a generalization of the consensus problem such that 1-set agreement is consensus. In the  $k$ -set agreement problem, each process proposes a value, and some processes eventually decide a value while respecting the properties of validity (a decided value is a proposed value), termination (every correct process eventually decides a value) and agreement (at most  $k$  values are decided).

Protocols solving consensus or  $k$ -set agreement have been proposed for dynamic systems, but they usually as-

sume synchronous communications (as in [1], [2], [3], [4]) or make strong assumptions on the number of process failures [5].

We approach the  $k$ -set agreement problem from a failure detector perspective [6], [7]. Failure detectors provide processes with information on process failures. They have been used as an abstraction of system assumptions to circumvent the impossibility of solving consensus in asynchronous systems prone to crash failures [8].

The  $\Pi\Sigma_{x,y}$  failure detector was introduced in [9] and is sufficient to solve  $k$ -set agreement in static networks (if and only if  $k \geq xy$ ) while being weaker than other known failure detectors which solve the same problem. However, this failure detector relies on information that is not available in unknown networks: the list of all the participating processes. Additionally, traditional failure detectors rely on a full connectivity of the network graph, which is not available in a dynamic network.

In the current paper we extend the definition of  $\Pi\Sigma_{x,y}$  in order to obtain the  $\Pi\Sigma_{\perp,x,y}$  failure detector, which is capable of solving  $k$ -set agreement in unknown dynamic systems, and provide implementations of this new detector. We also adapt the  $k$ -set agreement algorithm of [9], [10] to solve  $k$ -set agreement using  $\Pi\Sigma_{\perp,x,y}$  on top of our model.

The model assumptions we propose to implement  $\Pi\Sigma_{\perp,x,y}$  are generic and expressed in terms of message pattern, which allows our model to be applied to a range of systems. We also provide concrete examples of partial synchrony and failure pattern properties which are sufficient to ensure our generic assumptions.

The system is modeled using the formalism of the Time-Varying Graph (TVG), as defined in [11].

This paper thus brings the following main contributions:

- 1) The definition of the  $\Pi\Sigma_{\perp,x,y}$  failure detector as an adaptation of  $\Pi\Sigma_{x,y}$  to solve  $k$ -set agreement in unknown dynamic networks.

- D. Jeanneau, L. Arantes and P. Sens are with Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6
- T. Rieutord is with INFRES, Telecom ParisTech

D. Jeanneau was supported by the Labex SMART, supported by French state funds managed by the ANR within the Investissements d’Avenir programme under reference ANR-11-LABX-65.

T. Rieutord was supported by the ANR project DISCMAT, under grant agreement N ANR-14-CE35-0010-01.

- 2) An algorithm implementing  $\Pi\Sigma_{\perp,x,y}$  in our model, relying on connectivity and message pattern assumptions.
- 3) An algorithm solving  $k$ -set agreement in our model enriched with  $\Pi\Sigma_{\perp,x,y}$ , which is an adaptation of the  $k$ -set agreement algorithm for static connected networks presented in [9], [10].

The remaining of the paper is organized as follows: Section 2 formally describes our system model. Section 3 presents the definitions of several failure detectors relevant to our work, and introduces the  $\Pi\Sigma_{\perp,x,y}$  failure detector. Section 4 defines the different connectivity and message pattern assumptions that we rely on. In Section 5, we propose an implementation of  $\Pi\Sigma_{\perp,x,y}$ . Section 6 presents an algorithm solving  $k$ -set agreement with  $\Pi\Sigma_{\perp,x,y}$  for  $k \geq xy$ . Section 7 presents the related work. Finally, Section 8 concludes the paper.

## 2 SYSTEM MODEL

### 2.1 Process Model

A finite set of  $n$  processes  $\Pi = \{p_1, \dots, p_n\}$  participate in the system. *The processes are synchronous* (there is a bound on the relative speed of processes) and uniquely identified, although initially they are only aware of their own identities. Processes are not required to know the value of  $n$ .

A run is a sequence of steps executed by the processes while respecting the causality of operations (each received message has been previously sent). Processes can join and leave the system during the run ( $\Pi$  is the set of all processes that participate in the system at some point in time). Processes may also crash, and we make no difference between a process that crashes permanently and a process that leaves the system permanently: in both cases the process is considered *faulty* in that run. A process that is not faulty is called *correct*. Note that this definition of faulty and correct processes is not exactly the traditional one. Indeed, correct processes can crash or leave the system, as long as they recover or come back later. Only processes that crash or leave permanently are considered faulty.

Correct processes can leave the system and come back infinitely often, but they can only crash and recover a finite number of times. The critical difference is that a process that leaves keeps its memory intact, whereas a crashed process does not.

The set of all correct processes is called  $\mathcal{C}$ . We assume a bound  $f < n$  on the number of faulty processes in a run.

### 2.2 Communication Model

Processes communicate by sending and receiving messages. *Communications are asynchronous*: there is no bound on message transfer delays. Therefore, even though processes are synchronous, they do not cooperate in a synchronous way.

The system is dynamic, which means that nodes and communication links can appear or disappear during the run: therefore, the communication graph will change over time. The usual notion of path in the graph is not sufficient to define reachability in such a dynamic graph. To solve this issue, several solutions were proposed in the literature [1], [11], [12], [13]: we choose to model the communication graph using the Time-Varying Graph (TVG) formalism, as defined by Casteigts et al. in [11].

#### 2.2.1 Time-Varying Graphs

**Definition 1** (Time-Varying Graph). *A time-varying graph is a tuple  $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta, \psi)$  where*

- 1)  $V = \Pi$  is the set of nodes in the system.
- 2)  $E \subseteq V \times V$  is the set of edges.
- 3)  $\mathcal{T} = \mathbb{N}$  is a time span.
- 4)  $\rho : E \times \mathcal{T} \rightarrow \{0, 1\}$  is the edge presence function, indicating whether a given edge  $e \in E$  is active at a given time  $t \in \mathcal{T}$ .
- 5)  $\zeta : E \times \mathcal{T} \rightarrow \mathbb{N}$  is the latency function, indicating the time taken to cross an edge  $e \in E$  if starting at given time  $t \in \mathcal{T}$ .
- 6)  $\psi : V \times \mathcal{T} \rightarrow \{0, 1\}$  is the node presence function, indicating whether a given node  $p \in V$  is present in the system at a given time  $t \in \mathcal{T}$ . The edge presence function and the node presence function must be coherent:  $\forall t \in \mathcal{T}, \forall p_i \in V, \forall e \in E$ , if  $e$  is connected to  $p_i$  then  $\psi(p_i, t) = 0 \implies \rho(e, t) = 0$ .

$G(V, E)$  is the underlying graph of  $\mathcal{G}$ , and indicates which nodes have a relation at some time in  $\mathcal{T}$ .

Note that processes do not know the values of the  $\zeta$  function, which is only introduced for the simplicity of presentation. Since communications are asynchronous, the values of  $\zeta$  are finite but not necessarily bounded.

The communication links between processes are not permanent: the  $\rho$  function indicates when a given edge is active. Therefore, the usual notion of *path* in the graph is not suited to TVGs: journeys are defined for this purpose.

Intuitively, a journey is a path over time. In order to transmit a message from process  $p_i$  to process  $p_j$  in a TVG, it is not necessary for every edge on the path to be active at the time the message is sent: it is sufficient that there exists a path between  $p_i$  and  $p_j$  such that all the edges on the path are active in the right order at some time in the future.

**Definition 2** (Journey). *A journey is a sequence of couples  $\mathcal{J} = \{(e_1, t_1), (e_2, t_2), \dots, (e_m, t_m)\}$  such that  $\{e_1, e_2, \dots, e_m\}$  is a walk in  $G$  and:*

$$\forall i, 1 \leq i < m : (\rho(e_i, t_i) = 1) \wedge (t_{i+1} \geq t_i + \zeta(e_i, t_i)) .$$

$t_1$  is called *departure*( $\mathcal{J}$ ) and  $t_m + \zeta(e_m, t_m)$  is called *arrival*( $\mathcal{J}$ ). We denote  $\mathcal{J}_{(u,v)}^*$  the set of all the journeys starting at node  $u$  and ending at node  $v$ .

Consider the following example: a graph where  $E = V \times V$  and every edge in the system is active infinitely often (longer than the message transfer time), but no more than one edge is ever active at a time. In such a system, there are journeys infinitely often between every node and the connectivity is sufficient to solve complex problems such as consensus. However, at any given instant, the graph is partitioned into at least  $n - 1$  independent subsets. This shows that similarly to paths, the usual notion of graph partitioning loses relevancy in TVGs: the number of partitions at a particular instant in the run is not a very useful parameter. Instead we are interested in the number of *partitions over time*: in the rest of the paper, we use the word *partition* to refer to a subset of the network that is isolated from the rest of the network for an arbitrarily long duration, and not just temporarily.

#### 2.2.2 Communication primitive

Processes communicate exclusively by sending messages with a broadcast primitive. We are relying on a very simple

broadcast: when a process  $p_i$  calls the broadcast primitive, the message is simply sent to the processes that are currently in  $p_i$ 's neighborhood, including  $p_i$ . We do not require the broadcast to provide advanced features such as message forwarding, routing, message ordering or any guarantee of delivery.

### 2.2.3 Channels

The channels are fair-lossy. Messages may be lost but, if the edge is active for the entire time of the message transfer, a message sent infinitely often will be received infinitely often. Messages may be duplicated, but a message may only be duplicated a finite number of times. No message can be created or altered. We make no assumption on message ordering and do not require channels to be FIFO.

## 3 FAILURE DETECTORS

Failure detectors ([6]) are distributed oracles that provide processes with unreliable information on process failures, often in the form of a list of trusted process identities. This information is unreliable in the sense that the failure detector may erroneously consider a correct process as faulty, or vice versa, but will attempt to correct these mistakes later. Each failure detector class ensures some properties on the reliability of the failure information. A failure detector is an abstraction of the system assumptions used to solve a given problem.

The failure detector abstraction has been investigated as a way to circumvent the impossibility result of [8] and solve consensus in asynchronous systems prone to crash failures [7]. Our goal in this paper is to adapt this solution to solve  $k$ -set agreement in dynamic systems.

Traditionally, failure detectors are used in system models considering static and fully connected communication graphs. These connectivity properties are usually presented as properties of the system model rather than the failure detector augmenting it. When considering a much weaker system model such as a dynamic network, solving any non-trivial problem still requires the assumption of a certain degree of graph connectivity, as not much can be done in a system where no communication link is ever active. Studying dynamic systems means considering the level of temporal connectivity required to solve a specific problem: using a generic and strong connectivity assumption would defeat that purpose. Instead, the goal should be to use a weak connectivity assumption that is still sufficient to solve the problem. Therefore, to solve a given agreement problem, two things are necessary: (1) a failure detector and (2) a connectivity assumption.

But if connectivity assumptions must be added to the system model in addition to the failure detector, then it cannot be said that the failure detector is *sufficient* to solve the problem. For this reason, and because in a dynamic system the required level of connectivity is as dependent on the problem as the required failure detector, we consider that failure detectors for dynamic systems should include connectivity properties. Adding these connectivity properties should not be seen as strengthening the failure detectors: they are still weaker than the assumption of a fully connected, static communication graph.

Additionally, our system model considers an unknown network where processes have no information on system membership at the beginning of the run. A way to circumvent this issue was proposed in [14] in the form of the  $\Sigma_{\perp}$  failure detector

Our approach is based on the  $\Pi\Sigma_{x,y}$  failure detector of [9], augmented with connectivity properties and extended with the method of [14] in order to obtain a failure detector sufficient to solve the  $k$ -set agreement problem in unknown dynamic systems.

### 3.1 The Quorum Failure Detectors

The quorum failure detector  $\Sigma$ , [15], provides every process with sets of process identities (called quorums) such that any two quorums output by  $\Sigma$  at any time necessarily intersect. Additionally,  $\Sigma$  requires that all quorums eventually contain only correct processes.

The  $\Sigma_k$  failure detector, [16], is a generalization of  $\Sigma$  meant to solve  $k$ -set agreement. Similarly to  $\Sigma$ ,  $\Sigma_k$  provides processes with eventually correct quorums, and at least two out of any  $k + 1$  quorums must intersect. It is easy to see that  $\Sigma = \Sigma_1$ .

Intuitively,  $\Sigma_k$  prevents the network from partitioning into more than  $k$  independent subsets. Note that in the case of a dynamic network, this statement only applies to *partitions over time*: the network may still be instantaneously partitioned into any number of subsets at any given instant.

In message passing systems,  $\Sigma$  is necessary to solve consensus ([15]) and  $\Sigma_k$  is necessary to solve  $k$ -set agreement ([16]).

The intersection property of  $\Sigma$  and  $\Sigma_k$  must hold over time, which means that if a process queries its failure detector before any communication has taken place, the returned quorum must intersect with the quorums formed by other processes later in the run. In known networks, implementations of  $\Sigma$  traditionally solve this issue by returning  $\Pi$  as a quorum at the beginning of the run [15]. This is not an option in unknown networks where knowledge of the system membership is only established through communication.

The  $\Sigma_{\perp}$  failure detector, [14], is an adaptation of  $\Sigma$  for unknown networks: instead of returning a quorum,  $\Sigma_{\perp}$  can also output the default value  $\perp$  whenever the knowledge necessary to form a quorum has not been gathered yet.

In order to solve  $k$ -set agreement in unknown dynamic networks, we define the  $\Sigma_{\perp,k}$  failure detector, which combines the properties of  $\Sigma_k$  and  $\Sigma_{\perp}$ . It also includes a connectivity property which replaces (and is weaker than) the assumption of a static and complete network.

The  $\Sigma_{\perp,k}$  failure detector provides each process  $p_i$  with a quorum denoted  $qr_i^{\tau}$  (which is either a set of process identities or the special value  $\perp$ ) at any time instant  $\tau$ .

For the convenience of the presentation, we introduce the following definition:

**Definition 3** (Recurrent neighborhood). *The recurrent neighborhood of a correct process  $p_i$ , denoted  $R_i$ , is the set of all correct processes whose quorums intersect infinitely often with  $p_i$ 's quorums.  $\forall p_i \in \mathcal{C}, R_i = \{p_j \in \mathcal{C} \mid \forall \tau, \exists \tau_i, \tau_j \geq \tau : qr_i^{\tau_i} \neq \perp \wedge qr_j^{\tau_j} \neq \perp \wedge qr_i^{\tau_i} \cap qr_j^{\tau_j} \neq \emptyset\}$ .*

Note that  $p_j \in R_i$  is an equivalence relation between  $p_i$  and  $p_j$ . By definition,  $\forall p_i \in \mathcal{C} : p_i \in R_i$  and therefore  $R_i \neq \emptyset$ .

We say that a correct process  $p_i$  can *reach* another correct process  $p_j$  if, provided that  $p_i$  sends messages infinitely often,  $p_j$  receives them infinitely often.

$\Sigma_{\perp,k}$  is defined by the self-inclusion, quorum liveness, quorum intersection and quorum connectivity properties.

**Property 1** (Self-inclusion). *Every process includes itself in its non- $\perp$  quorums.  $\forall p_i \in \Pi, \forall \tau : (qr_i^\tau \neq \perp) \implies (p_i \in qr_i^\tau)$ .*

**Property 2** (Quorum liveness). *Eventually, every correct process stops returning  $\perp$  and its quorums only contain correct processes.  $\exists \tau, \forall p_i \in \mathcal{C}, \forall \tau' \geq \tau : qr_i^{\tau'} \neq \perp \wedge qr_i^{\tau'} \subseteq \mathcal{C}$ .*

**Property 3** (Quorum intersection). *Out of any  $k+1$  non- $\perp$  quorums, at least two intersect.*

$$\begin{aligned} & \forall \tau_1, \dots, \tau_{k+1} \in \mathcal{T}, \forall id_1, \dots, id_{k+1} \in \Pi, \\ & \exists i, j : 1 \leq i \neq j \leq k+1 : \\ & (qr_{id_i}^{\tau_i} \neq \perp \wedge qr_{id_j}^{\tau_j} \neq \perp) \implies (qr_{id_i}^{\tau_i} \cap qr_{id_j}^{\tau_j} \neq \emptyset) . \end{aligned}$$

**Property 4** (Quorum connectivity). *Every correct process  $p_i$  can reach every process in  $R_i$ .*

$\Sigma_k$  and  $\Sigma_{\perp}$  were defined with only 2 properties (liveness and intersection). Self-inclusion is a property added to  $\Sigma_x$  and  $\Pi\Sigma_x$  by the authors in [9] for the sake of the simplicity of algorithm proofs, and it is trivially implemented by the algorithms we present in the rest of the paper. Quorum connectivity is the property added to deal with the dynamicity of the network.

Intuitively, the quorum connectivity property means that processes belong to the same partition as their recurrent neighborhood. Note that  $\forall p_i, p_j \in \mathcal{C} : p_i \in R_j \implies p_j \in R_i$ , thus quorum connectivity enables two-way communication between  $p_i$  and  $p_j$ .

This property is not very costly, since most intuitive failure detector implementations would already require some level of connectivity between processes in a quorum in order to form the quorums. This is the case for the  $\Sigma_{\perp,k}$  algorithm we present in Section 5, which will not require any additional assumption to implement quorum connectivity.

### 3.2 The Family of Failure Detectors $\Pi\Sigma_{x,y}$

Although  $\Sigma_k$  is necessary to solve  $k$ -set agreement, it is not sufficient. It has been shown in [10] that  $k$ -set agreement can be solved in static asynchronous networks with  $\langle \Sigma_x, \bar{\Omega}_y \rangle$ , with  $k \geq xy$ , where  $\bar{\Omega}_y$  is the eventual anti-leader detector [17]. It was shown in the same paper that if  $n \geq 2xy$ , then there is no  $\langle \Sigma_x, \bar{\Omega}_y \rangle$ -based  $k$ -set algorithm for  $k < xy$ , which means that the  $k \geq xy$  requirement is tight.<sup>1</sup>

However in [9], Mostéfaoui, Raynal and Stainer introduce the  $\Pi\Sigma_{x,y}$  failure detector and prove that it is strictly weaker than  $\langle \Sigma_x, \bar{\Omega}_y \rangle$  for  $1 < y < x < n$  while still being strong enough to solve  $k$ -set agreement with  $k \geq xy$ . Interestingly,  $\Pi\Sigma_{x,y}$  is defined incrementally based on the properties of  $\Sigma_x$ . Therefore, an algorithm for  $\Sigma_x$  (or  $\Sigma_{\perp,x}$ , in our case) can easily be extended to implement  $\Pi\Sigma_{x,y}$  (resp.,  $\Pi\Sigma_{\perp,x,y}$ ), with an additional assumption.

The authors in [9] provide an intuitive description of  $\Pi\Sigma_{x,y}$ .  $\Pi\Sigma_{x,1}$  (1) prevents the system from partitioning into more than  $x$  partitions with the properties of  $\Sigma_x$  and

(2) guarantees that the processes of at least one of these subsets agree on a common leader.  $\Pi\Sigma_{x,y}$  can be seen as  $y$  independent instances of  $\Pi\Sigma_x$  in which (2) has to be guaranteed in only one of these instances.

We define  $\Pi\Sigma_{\perp,x,y}$  as an extension of  $\Pi\Sigma_{x,y}$  that includes the properties of  $\Sigma_{\perp,x}$  and is capable of solving  $k$ -set agreement in unknown dynamic systems.

### 3.3 The Family of Failure Detectors $\Pi\Sigma_{\perp,x,y}$

Similarly to [9],  $\Pi\Sigma_{\perp,x,y}$  is defined incrementally:  $\Pi\Sigma_{\perp,x}$  is defined firstly.

#### 3.3.1 The failure detector $\Pi\Sigma_{\perp,x}$

At any time instant  $\tau$ ,  $\Pi\Sigma_{\perp,x}$  provides each process  $p_i$  with a quorum denoted  $qr_i^\tau$  (which is either a set of process identities or the special value  $\perp$ ) and a leader denoted  $leader_i^\tau$  (which is a process identity).

$\Pi\Sigma_{\perp,x}$  is defined by the following properties:

- Self-inclusion
- Quorum liveness
- Quorum intersection
- Quorum connectivity
- Eventual partial leadership

First, we define an eventual partial leader as follows:

**Definition 4** (Eventual partial leader). *An eventual partial leader  $p_l$  is a correct process such that every process in the recurrent neighborhood of  $p_l$  eventually recognizes  $p_l$  as its leader forever.  $p_l \in \mathcal{C} \wedge \forall p_i \in R_l, \exists \tau, \forall \tau' \geq \tau : leader_i^{\tau'} = p_l$ .*

We denote  $L$  the set of all eventual partial leaders.

**Property 5** (Eventual partial leadership). *For every correct process  $p_i$ , there is an eventual partial leader  $p_l$  that can reach  $p_i$ .*

The original eventual partial leadership property used in [9] simply requires the existence of an eventual partial leader in the system. Our version of the property similarly implies that  $L \neq \emptyset$  (since  $\mathcal{C} \neq \emptyset$ ), but also implies that each correct process must be reachable by one eventual partial leader (which, depending on the level of connectivity, may require more than one leader). In a static and connected network, both properties are equivalent: a single eventual partial leader is necessary and sufficient to fulfill the property, since the connected communication graph enables this single leader to reach every correct process.

In a  $k$ -set agreement algorithm, the eventual partial leaders are the processes that are ensured to decide eventually. In order to ensure termination, the deciding leaders must, therefore, be able to inform the rest of the system of their decision. However, in a dynamic network, the mere existence of an eventual partial leader does not provide the latter with the necessary connectivity to ensure termination. This is why in dynamic networks, our eventual partial leadership property is stronger than the original one and imposes the required connectivity.

The eventual partial leadership property implies a trade-off between the number of eventual partial leaders in the system and graph connectivity. On one hand, if there is a single leader in the system, then this leader must be able to reach every correct process in the system. On the other

1. This result can also be proved using the impossibility of  $k$ -set agreement theorem ([18]), the premise of which applies if  $k < xy$ .

hand, if the communication graph is partitioned, then there must be at least one local leader per partition.

Such a trade-off implies that the eventual partial leadership property does not prevent the system from being partitioned into up to  $n$  partitions over time, provided that every correct process is its own eventual partial leader. However in this scenario it would be impossible to verify the quorum intersection and quorum connectivity properties.

### 3.3.2 The failure detector $\Pi\Sigma_{\perp,x,y}$

The definition of  $\Pi\Sigma_{\perp,x,y}$  is the same as  $\Pi\Sigma_{x,y}$  in [9], except that it uses  $\Pi\Sigma_{\perp,x}$  instead of  $\Pi\Sigma_x$ .  $\Pi\Sigma_{\perp,x,y}$  can be seen as  $y$  instances of  $\Pi\Sigma_{\perp,x}$  running concurrently.

$\Pi\Sigma_{\perp,x,y}$  provides each process  $p_i$  with an array  $FD_i[1..y]$  such that for each  $j$ ,  $1 \leq j \leq y$ ,  $FD_i[j]$  is a pair containing a quorum  $FD_i[j].qr$  and a process index  $FD_i[j].leader$ . The array satisfies the following properties:

**Property 6** (Vector safety).  $\forall j \in [1..y] : FD_i[j].qr$  satisfies the self-inclusion, liveness, intersection and quorum connectivity properties of  $\Pi\Sigma_{\perp,x}$ .

**Property 7** (Vector liveness).  $\exists j \in [1..y] : FD_i[j]$  satisfies the eventual partial leadership property of  $\Pi\Sigma_{\perp,x}$ .

The idea is to reduce the cost of the system assumptions: the liveness property only needs to be verified by one out of a set of  $y$  instances of the detector.

The authors in [9] prove that for  $1 \leq y \leq n - 1$ ,  $\Pi\Sigma_{1,y}$  is as strong as  $\langle \Sigma_1, \bar{\Omega}_y \rangle$ . This shows that the  $y$  parameter of  $\Pi\Sigma_{x,y}$  (and  $\Pi\Sigma_{\perp,x,y}$ ) is comparable to the  $y$  parameter of  $\bar{\Omega}_y$ .

$\Pi\Sigma_{\perp,x,y}$  is sufficient to solve the  $k$ -set agreement problem in our model if  $k \geq xy$ . We will prove this statement by providing a  $k$ -set agreement algorithm relying on  $\Pi\Sigma_{\perp,x,y}$  in Section 6.

## 4 ASSUMPTIONS

In this section we present some system assumptions. The algorithms presented in Section 5 will then list the assumptions from this section on which they rely.

### 4.1 Time-Varying Graph Classes

In addition to defining the formalism of the TVG, Casteigts et al. present in [11] a number of TVG classes which provide different levels of connectivity assumptions. We are particularly interested in class 5.

**Definition 5** (Class 5: recurrent connectivity [11]). *All processes can reach each other infinitely often through journeys.*  $\forall u, v \in \Pi, \forall \tau, \exists \mathcal{J} \in \mathcal{J}_{(u,v)}^* : \text{departure}(\mathcal{J}) > \tau$ .

This connectivity assumption does not exactly fit our purpose. On the one hand, it is too strong. It implies a global connectivity between any two processes in the system, which is not necessary to solve  $k$ -set agreement, since the problem can be solved in a system partitioned into  $k$  subsets.

On the other hand, class 5 is too weak since it relies on the notion of journey, which is insufficient to ensure the transmission of messages. Even if a journey exists between  $p_i$  and  $p_j$ , there is no guarantee that a message sent by  $p_i$  can reach  $p_j$ . In fact, even if the edge between  $p_i$  and  $p_j$

is active infinitely often and the message is sent infinitely often, the message might always be sent in between two activation periods of the edge, thus never crossing it. To solve this problem, Gómez-Calzado et al. defined in [19] the notion of timely journeys for the case of synchronous systems. We extend this solution into  $\gamma$ -journeys for the case of asynchronous communications.

**Definition 6** ( $\gamma$ -Journey). *A  $\gamma$ -journey  $\mathcal{J}$  (where  $\gamma > 0$  is a time duration) is a journey such that every node on the path can wait up to  $\gamma$  units of time after the next edge becomes active before forwarding the message. Since the message may be sent at any time during the  $\gamma$  time window and the channel latency may vary during that time, the edge must remain active long enough for the worst case duration.*

-  $\forall i, 1 \leq i \leq |\mathcal{J}|, e_i$  stays active from time  $t_i$  until, at least, time  $t_i + \max_{0 \leq j \leq \gamma} \{j + \zeta(e_i, t_i + j)\}$ .

-  $\forall i, 1 \leq i < |\mathcal{J}|, t_{i+1} \geq t_i + \max_{0 \leq j \leq \gamma} \{j + \zeta(e_i, t_i + j)\}$ .

With a  $\gamma$ -journey, processes are given an additional time window of  $\gamma$  units of time to send the message. In [19], this time was used to detect the activation of the edge. This solution is appropriate for point-to-point communications in a known network, since it allows the emitter of the message to resend the message to the receiver whenever the edge appears again. However this is not helpful in a unknown non-complete network where we have to rely on blind broadcasts and forwarding to propagate information.

Instead, we use the additional time window provided by  $\gamma$ -journeys as an upper bound on the time between two emissions of the message. This explains the need for synchronous processes: each process should be able to repeatedly send every message at least once every  $\gamma$  units of time.

Provided that processes receive their own broadcasts within  $\gamma$  time instants and then rebroadcast it, we can be sure that every message is sent at least once every  $\gamma$  instants.

If there is infinitely often a  $\gamma$ -journey from  $p_i$  to  $p_j$ , then  $p_i$  can reach  $p_j$ .

We call  $\mathcal{J}_{(u,v)}^\gamma$  the set of all the  $\gamma$ -journeys from  $u$  to  $v$ .

Using class 5 as a starting point, we define TVGs of class 5- $(\alpha, \gamma)$  as follows.  $\gamma$  is the time duration parameter of  $\gamma$ -journeys, and  $\alpha$  is a parameter defining how many processes each correct process is ensured to communicate with.

**Assumption 1** (Class 5- $(\alpha, \gamma)$ :  $(\alpha, \gamma)$ -recurrent connectivity). *Every correct process can reach and be reached through  $\gamma$ -journeys infinitely often by at least  $\alpha$  correct processes.*

$$\forall p_i \in \mathcal{C}, \exists P_i \subseteq \mathcal{C}, |P_i| \geq \alpha, \forall t \in \mathcal{T}, \forall p_j \in P_i,$$

$$\exists \mathcal{J}_i \in \mathcal{J}_{(p_i, p_j)}^\gamma : \text{departure}(\mathcal{J}_i) \geq t \wedge$$

$$\exists \mathcal{J}_j \in \mathcal{J}_{(p_j, p_i)}^\gamma : \text{departure}(\mathcal{J}_j) \geq t .$$

This assumption is parametrized by the two values  $\alpha$  and  $\gamma$ . A low  $\gamma$  value weakens the connectivity assumption by allowing shorter time windows for the journeys, but implies that processes must be able to send messages more often to ensure that a message is sent within the shorter window. On the other hand, a high  $\gamma$  value reduces the number of journeys that are qualified as  $\gamma$ -journeys, thus strengthening the connectivity assumption, but accepts slower processes.

The  $\alpha$  parameter also presents a trade-off: class 5- $(\alpha, \gamma)$  indirectly implies that there must be at least  $\alpha$  correct

processes in the system. As a result, a high  $\alpha$  value will result in a strong assumption on the number of process failures which can be costly in a dynamic system. A low  $\alpha$  value would strengthen the message pattern assumptions presented in the next section.

Class  $5-(\alpha, \gamma)$  also implies that all correct processes must know a lower bound of  $\alpha$ .

The assumption of a TVG of class  $5-(\alpha, \gamma)$  means that correct processes are able to communicate infinitely often with a subset of  $\alpha$  correct processes. This property ensures that correct processes will not wait for messages forever, which enables our algorithm to ensure the quorum liveness property. Additionally, if the algorithm ensures that every correct process  $p_i$  eventually only forms quorum from the  $P_i$  set, then class  $5-(\alpha, \gamma)$  also ensures quorum connectivity.

## 4.2 Message Pattern Assumptions

In this section we present message pattern assumptions, as defined by Mostéfaoui et al. in [20]. The message pattern model consists in assuming some properties on the relative order of message deliveries. If processes periodically wait for a certain number of messages, the idea is to assume that the message sent by some specific process will periodically be among the first ones received.

In order to express our message pattern assumptions, we will assume that the distributed algorithm executed by processes uses a query-response mechanism. Processes periodically issue query messages, to which other processes will respond.

The principle of our failure detector algorithm revolves around processes repeatedly issuing a query and then waiting for responses from  $\alpha$  processes:  $\alpha$  is therefore the minimum size of quorums returned by the algorithm. This does not necessarily constitute an assumption on the number of failures, since  $\alpha$  might be chosen equal to 1. This  $\alpha$  is the same parameter we used to define TVGs of class  $5-(\alpha, \gamma)$  which ensures that correct processes will not wait for messages for infinitely long.

We call *response set* the set of the first  $\alpha$  processes whose response to a given query issued by process  $p_i$  are received first by  $p_i$ .

### 4.2.1 Assumptions for Quorum Intersection

The assumption of a TVG of class  $5-(\alpha, \gamma)$  is not sufficient to ensure the quorum intersection property. In [10], Bouzid and Travers proposed a method to implement quorums: if processes repeatedly wait for messages from at least  $\lfloor \frac{n}{k+1} \rfloor + 1$  processes before using the new list of processes as their quorum, then the size of quorums alone is sufficient to ensure intersection. This method implies that there must be at least  $\lfloor \frac{n}{k+1} \rfloor + 1$  correct processes in the system, otherwise processes would wait forever, thus preventing liveness.

In a dynamic system where processes are expected to join and leave the system, such an assumption on the number of process failures seems too costly. For this reason we prefer to rely on the message pattern approach.

The following assumption is sufficient for our algorithm to implement quorum intersection. It was obtained by generalizing the assumption used for the case  $k = 1$  in [14].

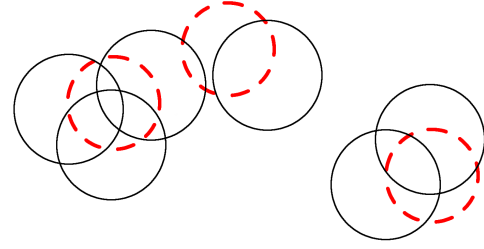


Fig. 1. Example of multiple winning quorums ( $m = k = 3$ ).

**Assumption 2** (Generalized winning quorums).  $\exists m \in [1, k]$  and  $\exists Q_{w1}, \dots, Q_{wm} \subseteq \Pi$  (called *winning quorums*). Each winning quorum  $Q_{wi}$  is associated with a number  $w_i \geq 1$  (called the *weight* of  $Q_{wi}$ ) such that  $\sum_{i=1}^m w_i \leq k$ .  $\forall p \in \Pi$ , every time  $p$  issues a new query,  $\exists i \in [1, m]$  such that  $Q_{wi} \neq \emptyset$  and out of the first  $\alpha$  processes from which  $p$  receives a response, at least  $\lfloor \frac{|Q_{wi}|}{w_i+1} \rfloor + 1$  of them are in  $Q_{wi}$ .

Intuitively, Assumption 2 requires that there are  $m$  sets of processes, the winning quorums, that answer faster than others, i.e., faster enough for parts of these sets to always be included in every response set. In addition, every time a correct process issues a query, connectivity must allow for part of one of these winning quorums to receive and respond to the query.

Note that winning quorums do not necessarily correspond to quorums returned by the failure detector at some point: instead they are sets of processes that have a tendency to be included in response sets.

The *weight*  $w_i$  of a winning quorum is a parameter which states which proportion of the winning quorum must be included in response sets. A winning quorum of weight 1 must be included in strict majority by a response set, whereas winning quorums of higher weights can be included in smaller proportions. The sum of all winning quorum weights is limited by  $k$ .

It is interesting to consider some extreme instances of this assumption. The first extreme is  $m = k$ : in this particular case, all winning quorums are necessarily of weight 1, and therefore each response set must include a strict majority of one of the winning quorums. Since each response set includes the strict majority of one out of  $k$  winning quorums, it is easy to see that out of any  $k + 1$  response sets, at least two will necessarily intersect.

Fig. 1 shows an example for  $m = k = 3$  in which winning quorums are represented by dashed red circles. Each solid black circle represents a response set. Note that out of any 4 response sets, at least 2 intersect.

Another extreme case is  $m = 1$  and  $w_1 = k$ . In this particular case, all response sets must contain a small part of a single winning quorum.

Fig. 2 shows an example for  $m = 1$  and  $k = 3$ . Similarly to Fig. 1, the winning quorum is represented by a dashed red circle, and response sets are represented by solid black circles. Once again, 2 out of any 4 response sets intersect.

The flexibility in the second example lies in which part of the winning quorum will be included in each response set, while the flexibility in the first example lied in which winning quorum would be included in majority by each response set.

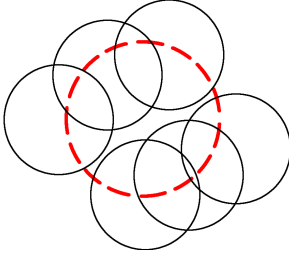


Fig. 2. Example of a single winning quorum ( $m = 1, k = 3$ ).

Assumption 2 implies that there is at least one winning quorum  $Q_{w_i}$  such that at least  $\lfloor \frac{|Q_{w_i}|}{w_i+1} \rfloor + 1$  of the processes in  $Q_{w_i}$  are correct. If  $\alpha = \lfloor \frac{|Q_{w_i}|}{w_i+1} \rfloor + 1 = 1$ , then there is no assumption on the number of failures but  $|Q_{w_i}| < w_i + 1$ , which only leaves minimal flexibility on the processes that must be included in every response set, thus strengthening the message pattern assumption. On the other hand if  $|Q_{w_i}|$  is high (and therefore  $\alpha$  is high), the number of failures is limited but each response set must contain a subset of a larger set, which allows for more flexibility in the message pattern.

#### 4.2.2 Assumption for Eventual Partial Leadership

In order to ensure the eventual partial leadership property, processes need to identify a local leader. Once again we choose to rely on a message pattern assumption. Since the eventual partial leadership property is supposed to be implemented on top of  $\Sigma_{\perp, x}$ , we can use the notion of quorum to define this new assumption.

Additionally, we use the order of processes in quorums to single out the leader. For this purpose we assume that processes in a quorum are totally ordered. Any specific ordering can be used: a natural choice would be to use the order in which the processes were added to the quorum. Another simple choice would be to order according to process identifiers. For a process  $p_i$  and a quorum  $qr$ , if  $p_i \in qr$ , then we denote by  $pos(p_i, qr)$  the position of  $p_i$  in  $qr$  according to the chosen total order. If  $p_i$  is the first process in  $qr$ , then  $pos(p_i, qr) = 1$ . In this particular case, we say that  $p_i$  is the *candidate* of  $p_j$  at time  $\tau$ .

We define our potential eventual partial leaders as follows:

**Definition 7** (Eventually winning process). *A correct process  $p_l$  is called an eventually winning process if there is a time  $\tau$  such that after  $\tau, \forall \tau' \geq \tau, \forall p_i \in R_l \setminus \{p_l\}$ :*

- 1)  $p_l$  is present in every quorum formed by  $p_i$ .  $p_l \in qr_i^{\tau'}$ .
- 2)  $p_l$ 's identity is always positioned in  $p_i$ 's quorum before the identities of other processes in  $R_i$ .  $\forall p_j \in R_i \setminus \{p_l, p_i\} : pos(p_l, qr_i^{\tau'}) < pos(p_j, qr_i^{\tau'})$ .
- 3) In every quorum formed by  $p_i$ , there is another process that also belongs to  $R_l$ .  $\exists p_j \in R_l \setminus \{p_l, p_i\} : p_j \in qr_i^{\tau'}$ .

Point (1) means that after some time,  $p_l$  must be fast enough for its responses to arrive in time to take part in every local quorum.

The implication behind (2) depends on the chosen ordering method. If processes are ordered by date of addition to the quorum, then (2) implies that after some time,  $p_l$  must be faster than the rest of the recurrent neighborhood of  $p_i$ .

If processes are ordered by process identities,  $p_l$  must have the smallest process identity in the recurrent neighborhood.

It is easy to see why (1) and (2) are useful: if  $p_l$  is part of every quorum and is singled out by the quorum order, then processes in  $R_l$  can reliably select their candidate as the leader.

Note that (2) excludes the case  $p_i = p_j$ , since otherwise  $p_l$  would have to be placed before  $p_i$  in  $p_i$ 's quorums. If processes are ordered by date of addition to the quorum, this expectation would be very unrealistic since receiving its own message is a local computation and should therefore be faster than receiving  $p_l$ 's message.

Point (3) requires that processes in  $R_l$  must not only communicate with  $p_l$  but also with each other to some extent, which enables them to share the information that  $p_l$  is their candidate.

Note that (3) also requires the processes  $p_l, p_i$  and  $p_j$  to be distinctly defined: therefore, in order for an eventually winning process to exist, there must be at least 3 correct processes in the system ( $f \leq n - 3$ ). Since  $p_l, p_i$  and  $p_j$  must be included in the same quorums,  $\alpha$  must also be equal to 3 or greater.

We call  $W$  the set of all eventually winning processes.

We can now formulate the assumption that will enable our failure detector algorithm to ensure the eventual partial leadership property:

**Assumption 3** (Eventually winning  $\gamma$ -sources). *For every correct process  $p_i$ , there is an eventually winning process  $p_l$  such that there is infinitely often a  $\gamma$ -journey from  $p_l$  to  $p_i$ .  $\forall p_i \in \mathcal{C}, \exists p_l \in W, \forall \tau : \exists \mathcal{J} \in \mathcal{J}_{(p_l, p_i)}^\gamma \wedge departure(\mathcal{J}) > \tau$ .*

Our  $\Pi\Sigma_{\perp, x}$  algorithm will ensure that eventually winning processes are eventual partial leaders. As a result, this assumption will be sufficient to ensure the eventual partial leadership property.

### 4.3 Summary of assumptions

Table 1 summarizes the assumptions presented in this section and the failure detector properties that they enable us to implement.

TABLE 1  
Assumptions for failure detector implementations

Assumption	Failure detector property
Assumption 1	$\Sigma_{\perp, k}$ : quorum liveness $\Sigma_{\perp, k}$ : quorum connectivity
Assumption 2	$\Sigma_{\perp, k}$ : quorum intersection
Assumption 3	$\Pi\Sigma_{\perp, x}$ : eventual partial leadership

The self-inclusion property of  $\Sigma_{\perp, k}$  is absent from this table because it does not require any assumption and will simply be ensured through algorithmic properties.

### 4.4 Implementation of Message Pattern Assumptions

Assumptions 2 and 3 are very abstract and it can be difficult to judge at first glance how likely they are of being verified in a real network. This is because we attempt to isolate assumptions that are specifically as close as possible to the minimum model strength required to ensure that our algorithm implements the  $\Pi\Sigma_{\perp, x, y}$  failure detector. The message



pattern model allows us to do this while keeping our model generic and applicable to different networks.

In this section we provide examples of more traditional assumptions that are sufficient to ensure Assumptions 2 and 3.

#### 4.4.1 Implementation of Assumption 2

A simple and intuitive method is to assume that  $|\mathcal{C}| \geq \lfloor \frac{n}{k+1} \rfloor + 1$ . In this case, Assumption 2 is trivially verified with  $m = 1, w_1 = k$  and  $Q_{w_1} = \Pi$ . This implies that  $\alpha \geq \lfloor \frac{n}{k+1} \rfloor + 1$ : the minimal size of quorums formed is then sufficient to ensure intersection. This particular case is the method used to implement  $\Sigma_k$  in static networks in [10].

Another method would be to use a partial synchrony assumption. For a given duration  $\Delta$ , let us call  $\Delta$ -journey a  $\gamma$ -journey  $\mathcal{J}$  such that  $arrival(\mathcal{J}) - departure(\mathcal{J}) \leq \Delta$ . We then separate  $\Pi$  into two subsets: slow processes and fast processes. A slow process  $p_i$  is a process such that there is never a  $\Delta$ -journey from  $p_i$  to any correct process  $p_j \in \mathcal{C} \setminus \{p_i\}$ . Fast processes are all other processes and  $Q$  is the set of all correct fast processes. The assumption is then that for any correct process  $p_i$  and any time instant, there are  $\Delta$ -journeys linking  $p_i$  to at least  $\lfloor \frac{|Q|}{k+1} \rfloor + 1$  processes from  $Q$ . Assumption 2 is then verified with  $m = 1, w_1 = k$  and  $Q_{w_1} = Q$ .

#### 4.4.2 Implementation of Assumption 3

One way to ensure Assumption 3 is that there is a correct subset  $Q$  of the system that is constantly connected and recognizes a leader  $p_l$ , that can reach the entire system infinitely often. The leader must be known from the other processes in  $Q$  from the start (it can simply be the lowest process identifier in  $Q$ , for example). When a process in  $Q$  issues a query, the communication layer for that process will then wait for a response from  $p_l$  and a response from another process in  $Q$  before delivering any other response. This is sufficient to ensure that  $p_l$  is the first process in every quorum formed in  $Q$ , and that processes in  $Q$  communicate with each other to a sufficient extent.

#### 4.4.3 Practical issues

From a practical point of view, some types of networks are particularly adapted to ensure Assumptions 2 and ???. In wireless mesh networks ([21]), the nodes move around a fixed set of nodes and each mobile node eventually connects to a fixed node. Wireless sensor networks ([22]) can be organized in clusters; one node in each cluster is designated the cluster head. Messages sent between clusters are routed through the cluster heads of the sending and receiving clusters. An infra-structured mobile network ([12]) is composed of Mobile Hosts (MH) and Mobile Support Stations (MSS). A MH is connected to a MSS if it is located in its transmission range, and two MHs can communicate only through MSSs.

In each of these network models, there is a privileged subset of powerful nodes (fixed nodes, cluster heads, MSSs) that can be used as a winning quorum to satisfy Assumption 2 or as the neighborhood  $R_l$  of an eventual winning process  $p_l$  for Assumption 3.

Both assumptions can also be ensured from a probabilistic perspective. If a subset  $Q$  of the system is made of powerful nodes that respond to queries much faster

than the rest of the nodes, then there is a high probability that Assumption 2 will be verified. Similarly, Assumption 3 can be verified in a probabilistic way with a leader that is simply a powerful process benefiting from very small communication delays with the processes around it.

## 5 FAILURE DETECTOR ALGORITHMS

In this section we first present a  $\Sigma_{\perp, k}$  algorithm, then extend it to obtain a  $\Pi\Sigma_{\perp, x, y}$  algorithm.

### 5.1 An Algorithm for $\Sigma_{\perp, k}$

Algorithm 1 implements the  $\Sigma_{\perp, k}$  failure detector in unknown dynamic systems with asynchronous communications. It uses a query/response mechanism with asynchronous round numbers in order to ensure quorum liveness.

#### 5.1.1 Assumptions

Algorithm 1 implements  $\Sigma_{\perp, k}$  in our model, provided that the following assumptions hold:

- 1) The system is a Time-Varying Graph of class 5- $(\alpha, \gamma)$  where  $\alpha$  is the minimal size of a quorum and  $\gamma$  is the maximal time taken by a process to receive its own broadcasts (Assumption 1).
- 2) The run follows a generalized winning quorums message pattern (Assumption 2).

**Algorithm 1.** Implementation of  $\Sigma_{\perp, k}$  for process  $p_i$ .

```

1: init
2:    $r_i \leftarrow 0$  //Local round number
3:    $qr_i \leftarrow \perp$  //The quorum returned by  $\Sigma_{\perp, k}$  for  $p_i$ 
4:    $recv\_from_i \leftarrow \{p_i\}$  //Quorum buffer
5:    $last\_known_i \leftarrow \emptyset$  //Round numbers of known processes
6:    $bcast(p_i, 0, \emptyset)$ 
7: end init
8:
9: task upon reception of  $(src, r\_src, Q)$  from  $p_j$ 
10:  if  $src = p_i$  and  $r\_src = r_i$  then //Response
11:     $recv\_from_i \leftarrow recv\_from_i \cup Q$ 
12:    if  $|recv\_from_i| \geq \alpha$  then
13:       $qr_i \leftarrow recv\_from_i$ 
14:       $recv\_from_i \leftarrow \{p_i\}$ 
15:       $r_i \leftarrow r_i + 1$ 
16:    end if
17:     $bcast(p_i, r_i, \emptyset)$ 
18:  else if  $src \neq p_i$  then //Query
19:    if  $\exists last\_r \mid (src, last\_r) \in last\_known_i$ 
20:       $\wedge last\_r \leq r\_src$  then
21:       $last\_known_i \leftarrow last\_known_i \setminus \{(src, last\_r)\}$ 
22:       $last\_known_i \leftarrow last\_known_i \cup \{(src, r\_src)\}$ 
23:       $bcast(src, r\_src, Q \cup \{p_i\})$ 
24:    else if  $(src, -) \notin last\_known_i$  then
25:       $last\_known_i \leftarrow last\_known_i \cup \{(src, r\_src)\}$ 
26:       $bcast(src, r\_src, Q \cup \{p_i\})$ 
27:    else
28:      do nothing
29:    end if
30:  end if
31: end task

```

#### 5.1.2 Notations

Each process  $p_i$  uses the following local variables:

- $r_i$  is the local round number of process  $p_i$ .
- $qr_i$  is the quorum currently returned by the failure detector for process  $p_i$ .

$recv\_from_i$  is the quorum buffer, containing all the identities of the processes whose message has been received by  $p_i$  since the time it last formed a new (complete) quorum. When the buffer gets big enough (at least  $\alpha$  process identities), it becomes the new quorum and  $recv\_from_i$  is reinitialized.

$last\_known_i$  is the knowledge  $p_i$  has of other processes round numbers. This variable and the associated mechanisms are not necessary for the correctness of the algorithm, they are simply used to improve performance by limiting the number of useless transmitted messages.

Process  $p_i$  calls the  $bcast(src, r\_src, Q)$  primitive to broadcast a message to all the processes currently in its immediate neighborhood. A message contains the following values:

$src$  is the identity of the original sender of the query (which is not necessarily the immediate sender of the message, since queries are forwarded multiple times).

$r\_src$  is the round number of  $src$  when this query was issued. Process  $src$  will ignore responses to previous rounds.

$Q$  is the set of the identities of processes who responded to this query so far. When the query goes back to process  $src$ , it will add the content of this set to its quorum buffer.

### 5.1.3 Algorithm Description

The principle behind the algorithm is the following: every process keeps broadcasting queries for round  $r_i$  until it receives enough responses to form a full quorum, then it increments  $r_i$  and proceeds with the next round.

Contrarily to most query/response algorithms, Algorithm 1 only uses one type of messages. A message is both a query and a response, depending on which process receives it. Every message travels from process to process, until it goes back to the original message sender. If the test on line 10 is true, the received message is considered as a response to the current round query. Otherwise if the test on line 18 is true, the message is considered as a query from another process.

Every process identity received in a response for the current round is added to the  $recv\_from_i$  buffer (line 11), and when the buffer size gets superior or equal to  $\alpha$ , then a new quorum is formed by copying  $recv\_from_i$  into  $qr_i$  and resetting the buffer (lines 12 – 15).

If a received message is a query from another process, then  $p_i$  updates its local knowledge then adds its own identity to the message and rebroadcasts it unless another query for a higher round has been previously received from the same emitter (lines 18 – 30).

At first glance it might look like process  $p_i$  only broadcasts its queries once (lines 6 and 17), but keep in mind that processes receive their own broadcasts: therefore after initially broadcasting a new query,  $p_i$  will receive it at most  $\gamma$  instants later and broadcast it again (line 17).

The same holds for queries from other processes. Once  $p_i$  has received a message from  $src$  for round  $r\_src$ , it will keep rebroadcasting it (lines 23 and 26) until it is informed that  $src$  moved on past round  $r\_src$  (the test on lines 19 – 20).

Based on the assumption of generalized winning quorums, the only action necessary to ensure quorum intersec-

tion is to make sure that quorums are formed from at least  $\alpha$  process identities, which is guaranteed by line 12.

Quorum liveness is ensured because (1) correct processes keep forming new quorums from fresh information infinitely often thanks to class 5- $(\alpha, \gamma)$  and (2) the identities of crashed processes are excluded from new quorums since the round numbers in their responses are eventually outdated (line 10).

### 5.1.4 Proof of Correctness

**Lemma 1.** *In a TVG of class 5- $(\alpha, \gamma)$  where Assumption 2 holds, Algorithm 1 ensures the quorum intersection property of  $\Sigma_{\perp, k}$ .*

*Proof:* Assumption 2 implies that  $\sum_{i=1}^m w_i \leq k$ . For any number  $w \in [1, k]$ , we denote  $n_w$  the number of winning quorums of weight  $w$ . It follows that  $\sum_{w=1}^k w \times n_w \leq k$ .

Additionally, Assumption 2 imposes that every response set is formed from a winning quorum  $Q_{w_i}$  of weight  $w_i$  such that at least  $\lfloor \frac{|Q_{w_i}|}{w_i+1} \rfloor + 1$  processes from  $Q_{w_i}$  are part of that response set. It follows that if  $w_i + 1$  response sets are formed from the same winning quorum  $Q_{w_i}$ , at least two of these response sets intersect.

If no two response sets are to intersect, then at most  $w_i$  response sets can be formed from a given winning quorum  $Q_{w_i}$ . Therefore, for any number  $w \in [1, k]$ , at most  $w \times n_w$  response sets can be formed from the set of all winning quorums of weight  $w$ . It follows finally that at most  $\sum_{w=1}^k w \times n_w$  response sets can be formed from the set of all winning quorums without any two of them intersecting. Since  $\sum_{w=1}^k w \times n_w \leq k$ , at least two out of any  $k + 1$  response sets intersect.

Lines 12 and 13 of Algorithm 1 ensure that quorums include the first  $\alpha$  responses (response set) received for the current query. Therefore every quorum includes a response set, and the quorum intersection property of  $\Sigma_{\perp, k}$  is ensured.  $\square$

**Lemma 2.** *In a TVG of class 5- $(\alpha, \gamma)$ , every correct process executing Algorithm 1 forms a new quorum infinitely often.*

*Proof:* Since it uses a query-response mechanism, Algorithm 1 requires every correct process to reach and be reached back by  $\alpha$  processes. A TVG of class 5- $(\alpha, \gamma)$  ensures that this happens infinitely often. Even if a journey includes waiting time during which the process holding the message is isolated, the process keeps memory of the message by rebroadcasting it to itself, and transmits it to other processes as soon as it stops being isolated. As a result, every correct process will receive responses from  $\alpha$  processes infinitely often, and therefore pass the test on line 12 infinitely often.  $\square$

**Lemma 3.** *In a TVG of class 5- $(\alpha, \gamma)$ , Algorithm 1 ensures the quorum liveness property of  $\Sigma_{\perp, k}$ .*

*Proof:* By definition, faulty processes will crash or leave the system forever in a finite time. Let  $t \in \mathcal{T}$  be the time at which the last faulty process crashes or leaves the system forever. Since  $f < n$ , there are correct processes in the system. Lemma 2 ensures that each of these processes will form a new quorum sometime after  $t$ . Let  $\tau \in \mathcal{T}$  be a time such that  $\tau > t$  and every remaining process has

formed a *quorum* between  $t$  and  $\tau$ . Therefore, every quorum being currently built at  $\tau$  has been started after  $t$ , which means no faulty process can possibly respond to the corresponding query message. As a result, every new quorum formed after  $\tau$  contains only correct processes. It follows that Algorithm 1 ensures the quorum liveness property of  $\Sigma_{\perp,k}$ .  $\square$

**Lemma 4.** *In a TVG of class 5- $(\alpha, \gamma)$ , Algorithm 1 ensures the quorum connectivity property of  $\Sigma_{\perp,k}$ .*

*Proof:* The properties of a TVG of class 5- $(\alpha, \gamma)$  ensure that every correct process will always receive enough messages to pass the test on line 12 and keep forming new quorums infinitely often. The test on line 10 ensures that processes only form quorums from messages from the current round. It follows that eventually, every correct process  $p_i$  only includes in its quorums processes which receive its queries and respond to it infinitely often. Therefore  $p_i$  can send and receive messages infinitely often to and from the processes that are infinitely often in its quorums.

Let  $p_i \in \mathcal{C}$  and  $p_j \in R_i$ . By definition of  $R_i$ ,  $p_i$  and  $p_j$ 's quorums intersect infinitely often and thus there must exist a correct process  $p_m$  such that  $p_m$  is infinitely often in  $p_i$ 's quorums and  $p_m$  is infinitely often in  $p_j$ 's quorums. As a result,  $p_m$  can receive messages from  $p_i$  infinitely often and  $p_j$  can receive messages from  $p_m$  infinitely often. Therefore if messages are routed through  $p_m$ ,  $p_j$  can receive messages from  $p_i$  infinitely often.  $\square$

**Theorem 1.** *In a TVG of class 5- $(\alpha, \gamma)$  where Assumption 2 holds, Algorithm 1 implements a  $\Sigma_{\perp,k}$  failure detector.*

*Proof:* It follows from Lemmas 1, 3 and 4 that the algorithm ensures the quorum intersection, quorum liveness and quorum connectivity properties.

Self-inclusion is ensured by the fact that every quorum is formed from the buffer  $recv\_from_i$  (line 13), and the buffer is always initialized with  $p_i$  (lines 4 and 14).  $\square$

## 5.2 An Algorithm for $\Pi\Sigma_{\perp,x}$

Algorithm 2 is an extension of Algorithm 1 meant to implement  $\Pi\Sigma_{\perp,x}$  in our dynamic model. It adds an election mechanism to the original algorithm in order to identify an eventual partial leader.

This leader election mechanism relies on the quorum order, as defined in Section 4.2.2. Every time a process forms a new quorum, it selects the first process in the quorum as candidate for the leader election. If a process is the candidate of every other process in its quorum, then it selects itself as leader; otherwise it selects its candidate as leader.

### 5.2.1 Assumptions

Algorithm 2 implements  $\Pi\Sigma_{\perp,x}$  in our model, provided that the following assumptions hold:

- 1) The system is a Time-Varying Graph of class 5- $(\alpha, \gamma)$  where  $\alpha$  is the minimal size of a quorum and  $\gamma$  is the maximal time taken by a process to receive its own broadcasts (Assumption 1).
- 2) The run follows a generalized winning quorums message pattern (Assumption 2).
- 3) The system verifies the eventually winning  $\gamma$ -sources

assumption (Assumption 3). As a consequence of this assumption,  $\alpha \geq |\mathcal{C}| \geq 3$ .

**Algorithm 2.** *Implementation of  $\Pi\Sigma_{\perp,x}$  for process  $p_i$ .*

```

1: init
2:    $r_i \leftarrow 0$  //Local round number
3:    $qr_i \leftarrow \perp$  //The quorum returned by  $\Pi\Sigma_{\perp,x}$  for  $p_i$ 
4:    $recv\_from_i \leftarrow \{p_i\}$  //Quorum buffer
5:    $last\_known_i \leftarrow \emptyset$  //Round numbers of known processes
6:    $leader_i \leftarrow p_i$  //The leader returned by  $\Pi\Sigma_{\perp,x}$  for  $p_i$ 
7:    $candidate_i \leftarrow \perp$  //  $p_i$ 's current candidate for leadership
8:    $candidates_i \leftarrow \emptyset$  //Candidates of processes in  $recv\_from_i$ 
9:    $bcast(p_i, 0, \emptyset, \emptyset)$ 
10: end init
11:
12: task upon reception of  $(src, r\_src, Q, cands)$  from  $p_j$ 
13:   if  $src = p_i$  and  $r\_src = r_i$  then //Response
14:      $recv\_from_i \leftarrow recv\_from_i \cup Q$ 
15:      $candidates_i \leftarrow candidates_i \cup cands$ 
16:     if  $|recv\_from_i| \geq \alpha$  then
17:        $qr_i \leftarrow recv\_from_i$ 
18:        $recv\_from_i \leftarrow \{p_i\}$ 
19:        $r_i \leftarrow r_i + 1$ 
20:        $candidate_i \leftarrow p_l \mid (pos(p_l, qr_i) = 1 \wedge p_l \neq p_i)$ 
21:          $\vee (pos(p_i, qr_i) = 1 \wedge pos(p_l, qr_i) = 2)$ 
22:       if  $candidates_i = \{p_i\}$  or  $\emptyset$  then
23:          $leader_i \leftarrow p_i$ 
24:       else
25:          $leader_i \leftarrow candidate_i$ 
26:       end if
27:        $candidates_i \leftarrow \emptyset$ 
28:     end if
29:      $bcast(p_i, r_i, \emptyset, \emptyset)$ 
30:   else if  $src \neq p_i$  then //Query
31:     if  $\exists last\_r \mid (src, last\_r) \in last\_known_i$ 
32:        $\wedge last\_r \leq r\_src$  then
33:          $last\_known_i \leftarrow last\_known_i \setminus \{(src, last\_r)\}$ 
34:          $last\_known_i \leftarrow last\_known_i \cup \{(src, r\_src)\}$ 
35:          $bcast(src, r\_src, Q \cup \{p_i\}, cands \cup \{candidate_i\})$ 
36:       else if  $(src, -) \notin last\_known_i$  then
37:          $last\_known_i \leftarrow last\_known_i \cup \{(src, r\_src)\}$ 
38:          $bcast(src, r\_src, Q \cup \{p_i\}, cands \cup \{candidate_i\})$ 
39:       else
40:         do nothing
41:       end if
42:     end if
43:   end task

```

### 5.2.2 Notations

Algorithm 2 uses the same notations as Algorithm 1. Additionally, each process  $p_i$  uses the following local variables:

$leader_i$  is the leader returned by the failure detector for process  $p_i$ .  $leader_i$  is initially  $p_i$ , and is later updated on lines 23 or 25.

$candidate_i$  is the first process in  $p_i$ 's most recent quorum (excluding  $p_i$  itself). It is defined by the variable affectation in lines 20 – 21.  $candidate_i$  is initialized to  $\perp$  and is added to sets (lines 35 and 38). We take the convention that  $\emptyset \cup \{\perp\} = \emptyset$ .

$candidates_i$  is the set of the candidates of the processes in  $recv\_from_i$  (except  $p_i$ ).  $p_i$  will only elect itself as leader (line 23) if  $candidates_i$  only contains  $p_i$  (meaning that  $p_i$  is the candidate of every process in  $recv\_from_i \setminus \{p_i\}$ ) or if  $candidates_i$  is empty (meaning that  $p_i$  considers itself alone).

In addition to the message parameters described for Algorithm 1, messages sent by processes contain the  $cands$

parameter, which is the set of the candidates of the processes in  $Q$ , at the time when they responded to the query. It carries the information necessary for process  $p_i$  to build its  $candidate_i$  set on line 15.

### 5.2.3 Algorithm Description

Algorithm 2 is an extension of Algorithm 1. It follows the same structure and uses the same mechanisms to build quorums. Its additional code aims to allow for the selection of a partial leader according to the eventual partial leadership property of  $\Pi\Sigma_{\perp,x}$ .

The algorithm is composed of two parts: candidate selection and leader selection.

Candidate selection revolves around the notion of quorum order presented in Section 4.2.2. The first process in every quorum is selected as the candidate. Whenever a process  $p_i$  completes a new quorum (meaning it passes the test on line 16), it handles the end of the round similarly to Algorithm 1 (lines 17 – 19). It then identifies the first process in the new quorum (excluding itself) according to the chosen ordering method in lines 20 – 21 and selects it as its  $candidate_i$ . If it was possible for  $p_i$  to be its own candidate, and if quorums were ordered by date of response, then  $p_i$  would always be its own candidate.

By virtue of Definition 7, an eventually winning process  $p_l$  will eventually be forever the candidate of every process in  $R_l \setminus \{p_l\}$ . However  $p_l$  cannot be its own candidate. Therefore, information about  $p_l$ 's own quorum order is not sufficient for  $p_l$  to select itself as the leader. It must take into account the candidates of other processes.

This is the purpose of the  $candidate_i$  variable. Other processes inform  $p_i$  of their respective candidates by including it in their responses (lines 35 and 38), and  $p_i$  gathers this information in  $candidate_i$  in line 15. When  $p_i$  completes a quorum,  $candidate_i$  contains the candidates of the processes currently in  $qr_i \setminus \{p_i\}$ .

If every process in  $qr_i$  agrees on  $p_i$  as the candidate (or if  $p_i$  is the only process in  $qr_i$ ), then  $p_i$  selects itself as the leader (line 23). Otherwise,  $p_i$  selects  $candidate_i$  (line 25).

Note that point (3) of Definition 7 prevents the problematic case where a process  $p_i$  only includes in its quorums an eventually winning process  $p_l$  and processes in  $\Pi \setminus R_l$ . In this case it would be possible for every process in  $R_i$  (including  $p_l$ ) to have  $p_i$  as their candidate infinitely often, thus misleading  $p_i$  into selecting itself as the leader infinitely often.

### 5.2.4 Proof of Correctness

We should prove that, if Assumptions 1, 2 and 3 hold, then Algorithm 2 ensures the 5 properties of  $\Pi\Sigma_{\perp,x}$ .

**Lemma 5.** *In a TVG of class 5- $(\alpha, \gamma)$  where Assumption 2 holds, Algorithm 2 ensures the self-inclusion, quorum intersection, quorum liveness and quorum connectivity properties of  $\Pi\Sigma_{\perp,x}$ .*

*Proof.* The added code from Algorithm 1 does not modify the way the  $qr_i$  variable is initialized and updated, therefore the proof for Theorem 1 holds for Algorithm 2.  $\square$

**Lemma 6.** *Every eventually winning process  $p_l$  is eventually forever the  $candidate_i$  of every process  $p_i (\neq p_l)$  of its recurrent neighborhood.  $\forall p_l \in W, \forall p_i \in R_l \setminus \{p_l\} : \exists \tau : \forall \tau' \geq \tau : candidate_i = p_l$  at time  $\tau'$ .*

*Proof.* It follows from the properties of a TVG of class 5- $(\alpha, \gamma)$  that correct processes will keep passing the test on line 16, and therefore will form new quorums infinitely often.

By contradiction, let us assume the following:  $\exists p_l \in W, \exists p_i \in R_l \setminus \{p_l\}, \exists p_m \in \Pi \setminus \{p_l\}, \forall \tau : \exists \tau' \geq \tau : candidate_i = p_m$  at time  $\tau'$ . There are, thus, two cases:

$p_m \notin R_i$ . By definition of  $R_i$ , there is a time after which  $p_i$ 's quorums never intersect with  $p_m$ 's quorum. By construction of the algorithm (lines 4 and 18), self-inclusion is ensured (every process belongs to its own quorums). Thus, there is a time after which  $p_m$  is never in  $p_i$ 's quorums, and therefore it can never be selected as  $candidate_i$  on lines 20 – 21 after this time.

$p_m \in R_i$ . Since  $p_l$  is an eventually winning process, there is a time after which (1)  $p_l$  is in every quorum formed by  $p_i$  and (2) in every quorum formed by  $p_i$  that includes  $p_m$ ,  $p_l$  is positioned before  $p_m$ . As a result,  $p_m$  can never be selected as  $candidate_i$  on lines 20 – 21 after this time.  $\square$

**Lemma 7.** *Every eventually winning process is an eventual partial leader.  $W \subseteq L$ .*

*Proof.* Let  $p_l \in W$ .  $p_l$  is an eventual partial leader if and only if, for every  $p_i \in R_l$ , eventually  $leader_i = p_l$  forever. There are two cases:

$p_i = p_l$ . It follows from the definition of  $R_l$  and from self-inclusion that there is a time after which every process that is not in  $R_l$  will stop appearing in the quorums formed by  $p_l$ . It follows that there is a time  $\tau_1$  such that  $\forall \tau'_1 > \tau_1, qr_l^{\tau'_1} \subseteq R_l$ . If  $\alpha = 1$ , then  $qr_l^{\tau'_1} = \{p_l\}$  and therefore  $candidate_l = \emptyset$  at time  $\tau'_1$  (by construction of  $candidate_l$ ). If  $\alpha > 1$ , since the definition of  $R_l$  is symmetrical,  $\forall \tau'_1 > \tau_1, \forall p_j \in qr_l^{\tau'_1} : p_l \in R_j$ . It then follows from Lemma 6 that  $\exists \tau_2 \geq \tau_1, \forall \tau'_2 > \tau_2, \forall p_j \neq p_l \in qr_l^{\tau'_2} : candidate_j = p_l$  at time  $\tau'_2$ . Since  $p_l$  will keep forming new quorums with fresh information,  $\exists \tau_3 \geq \tau_2$  such that every time after  $\tau_3$  that  $p_l$  completes a round, then  $candidate_l = \{p_l\}$ . As a result, after time  $\tau_3$ ,  $p_l$  will always pass the test on line 22 and therefore will forever identify itself as the leader.

$p_i \neq p_l$ . According to point (3) of the eventually winning process definition,  $\exists \tau_1, \forall \tau'_1 > \tau_1, \exists p_j \in R_l : p_j \in qr_i^{\tau'_1}$ . It follows from Lemma 6 that  $\exists \tau_2 \geq \tau_1, \forall \tau'_2 > \tau_2, candidate_j = candidate_i = p_l$  at time  $\tau'_2$ . Since  $p_i$  will keep forming new quorums with fresh information received from  $p_j$ ,  $\exists \tau_3 \geq \tau_2$  such that every time after  $\tau_3$  that  $p_l$  completes a round, then  $p_l \in candidate_i$ . As a result, after  $\tau_3$ ,  $p_i$  will always fail the test on line 22 and therefore will forever identify  $candidate_i = p_l$  as the leader. In both cases,  $p_i$  selects  $p_l$  as leader forever, which makes  $p_l$  an eventual partial leader.  $\square$

**Lemma 8.** *If the eventually winning  $\gamma$ -sources assumption holds, then Algorithm 2 ensures the eventual partial leadership property of  $\Pi\Sigma_{\perp,x}$ .*

*Proof.* It follows from Assumption 3 that  $\forall p_i \in \mathcal{C}, \exists p_l \in W, \forall \tau : \exists \mathcal{J} \in \mathcal{J}_{(p_i, p_l)}^{\gamma} \wedge departure(\mathcal{J}) > \tau$ . It follows from Lemma 7 that  $p_l \in L$ . Since we assume fair-lossy channels, then if  $p_l$  sends messages infinitely

often, then  $p_i$  will receive messages from  $p_l$  infinitely often.  $\square$

**Theorem 2.** *In a TVG of class  $5-(\alpha, \gamma)$  where Assumptions 2 and 3 hold, Algorithm 2 implements a  $\Pi\Sigma_{\perp, x}$  failure detector.*

*Proof:* The proof follows directly from Lemmas 5 and 8.  $\square$

### 5.3 An Algorithm for $\Pi\Sigma_{\perp, x, y}$

An algorithm for  $\Pi\Sigma_{\perp, x, y}$  simply consists in executing  $y$  instances of Algorithm 2 simultaneously. This algorithm relies on Assumptions 1 and 2 as well as on the eventually winning  $\gamma$ -sources assumption. However, this last assumption is only required to apply for one out of the  $y$  instances of the algorithm.

## 6 A $k$ -SET AGREEMENT ALGORITHM

In [9], the authors proposed an algorithm for  $k$ -set agreement using  $\Pi\Sigma_{x, y}$  for static networks. The  $k$ -set algorithm itself is very simple. It only deals with the liveness property of  $k$ -set agreement (termination) and encapsulates the safety properties (validity and agreement) into the  $Alpha_x$  sub protocol. In this section we will adapt the  $Alpha_x$  and  $k$ -set agreement algorithms for dynamic networks.

### 6.1 The $Alpha_x$ Sub Protocol

$Alpha$  was introduced in [23] as a way to exactly capture the safety properties of consensus (that is, validity and agreement). It is thus complementary to the  $\Omega$  failure detector, which is necessary to ensure liveness (the termination property). It was later generalized into  $Alpha_x$  for the  $x$ -set agreement problem in [24].

$Alpha_x$  is a distributed object used to store values proposed by processes. It initially stores the default value  $\perp$ . It provides processes with an operation  $propose(r, v)$  that returns a value (possibly  $\perp$ ). The round number  $r$  is a logical time and  $v$  is a proposed value. It is assumed that (a) each process will use increasing round numbers in successive invocations of  $propose()$  and (b) distinct processes use different round numbers. An  $Alpha_x$  object is defined by the following properties:

- **Termination.** Any invocation of  $propose()$  by a correct process terminates.
- **Validity.** If  $propose(r, v)$  returns  $v' \neq \perp$ , then  $propose(r', v')$  has been invoked with  $r' \leq r$ .
- **Quasi-agreement.** At most  $x$  different non- $\perp$  values can be returned by different  $propose()$  invocations.
- **Obligation.** Let  $p_l$  be a correct process and  $Q(l, \tau) = \{p_i \in \mathcal{C} \mid \forall \tau_i, \tau_i \geq \tau : qr_i^{\tau_i} \cap qr_l^{\tau_i} = \emptyset\}$ . If, after time  $\tau$ , (a) only  $p_l$  and processes of  $Q(l, \tau)$  invoke  $propose()$  and (b)  $p_l$  invokes  $propose()$  infinitely often, then at least one invocation issued by  $p_l$  returns a non- $\perp$  value.

Note that the termination property of  $Alpha_x$  is not related to the termination property of the  $k$ -set agreement.

In order to ensure the safety properties of  $k$ -set agreement, it is not necessary to make use of the eventual partial leadership property of  $\Pi\Sigma_{\perp, x}$  and therefore, the  $Alpha_x$  algorithm presented in the appendix does not make use of the

$leader_i$  variable. However, the  $k$ -set agreement algorithm implements the termination property of  $k$ -set agreement by relying on the obligation property of  $Alpha_x$  and the eventual partial leadership property of  $\Pi\Sigma_{\perp, x}$ .

The definitions in [10] and [9], that propose  $k$ -set agreement algorithms for static networks, use different obligation properties. The  $Alpha_x$  we use in this paper is the one defined in [9], which is weaker than the one in [10] by being  $\Sigma_x$ -aware.

### 6.2 $Alpha_x$ Algorithm

In this section we propose an algorithm implementing  $Alpha_x$  for unknown dynamic networks, adapted from the algorithm in [9].

Implementations of  $Alpha_x$  on top of  $\Sigma_x$  are provided in [10] and [9] (the latter being an improvement of the former, accounting for the different obligation property definition). Algorithm 3 is a variant of the algorithm of [9] adapted to dynamic networks. It implements  $Alpha_x$  in our model enriched with  $\Pi\Sigma_{\perp, x}$ .

The algorithm gives each proposed value a priority. Each process  $p_i$  keeps a value  $est_i$ , which is its current estimation of the value it will decide, and a pair  $(lre_i, pos_i)$  which defines the priority of value  $est_i$ .  $lre_i$  is the highest round seen by  $p_i$  and  $pos_i$  is the position of value  $est_i$  within round  $lre_i$ . The position is used to fix priority on proposed values.

The function  $g(\rho, \delta) = 2^\delta(\rho - 1) + 1$  where  $\rho$  is the position of value  $v$  on round  $r$  and  $\delta = r' - r$ , with  $r' \geq r$ , is used to compute the position of  $v$  on round  $r'$ .

If value  $v$  has priority  $\rho$  at round  $r$  and value  $v'$  has priority  $\rho'$  at round  $r'$  with  $r \leq r'$ ,  $v$  has lower priority than  $v'$  at round  $r'$  if and only if  $g(\rho, r' - r) < \rho'$  or  $(g(\rho, r' - r) = \rho') \wedge (v < v')$ .

The  $propose()$  function is composed of two phases. In the read phase (lines 8 – 16), the process attempts to gather knowledge on the values already proposed by other processes in a quorum (as defined by  $\Sigma_{\perp, x}$ ) by sending  $REQ\_R$  messages and receiving  $RSP\_R$  messages. If a process in the quorum is already computing a higher round,  $p_i$  returns  $\perp$  (line 13). Otherwise it selects the highest priority value it knows of (lines 14 – 15), and proceeds to the write phase.

In the write phase (lines 17 – 28), the process attempts to raise the priority of its current estimated value by communicating it to other processes in a quorum with  $REQ\_W$  messages and receiving  $RSP\_W$  messages. Once again, if any process in the quorum is computing a higher round,  $p_i$  returns  $\perp$  (line 25). If another process has a value of higher priority for the current round,  $p_i$  adopts it as its new estimated value (lines 26 – 27).  $p_i$  then raises  $pos_i$  by 1 (line 18) and repeats the write phase until it manages to raise a value to position  $2^r$  (line 17) or until it encounters a process in a higher round (line 25).

The following modifications were made from [9] in order for the algorithm to ensure the properties of  $Alpha_x$  in dynamic networks:

The original algorithm assumed a complete, static communication graph with reliable channels and therefore every message was only sent once. In our model we need messages to be rebroadcast (lines 38, 42, 55 and 59). This mechanism ensures that (1) the emitting process will rebroadcast its own message every  $\gamma$  time instants; and (2) the reception

of the message will not be restricted to the neighbors of the emitting process. The message will be received by every process that can be reached through a  $\gamma$ -journey.

Since messages are rebroadcast, the direct emitter of a message is not necessarily the source of the message. For this reason, we added the process identifier of the responding process in message types RSP\_R and RSP\_W.

**Algorithm 3.** Implementation of  $Alpha_x$  using  $\Pi\Sigma_{\perp,x}$  in dynamic networks for process  $p_i$ .

```

1: init
2:    $lre_i \leftarrow 0$ 
3:    $est_i \leftarrow \perp$ 
4:    $pos_i \leftarrow 0$ 
5: end init
6:
7: function PROPOSE( $r, v_i$ )
8:   repeat  $Q_i \leftarrow qr_i$ ; broadcast REQ_R( $r, Q_i$ )
9:   until  $Q_i \neq \perp$  and  $\forall p_j \in Q_i$  :
10:    RSP_R( $r, p_j, \langle lre_j, pos_j, val_j \rangle$ ) received
11:     $rcv_i \leftarrow \{ \langle lre_j, pos_j, est_j \rangle : p_j \in Q_i \wedge$ 
12:      RSP_R( $r, p_j, \langle lre_j, pos_j, est_j \rangle$ ) received }
13:    if  $\exists \langle lre, -, - \rangle \in rcv_i : lre > lre_i$  then return( $\perp$ ) end if
14:     $pos_i \leftarrow \max\{pos \mid \langle r, pos, v \rangle \in rcv_i\}$ 
15:     $est_i \leftarrow \max\{v \mid \langle r, pos_i, v \rangle \in rcv_i\}$ 
16:    if  $est_i = \perp$  then  $est_i \leftarrow v_i$  end if
17:    while  $pos_i < 2^r$  do
18:       $pos_i \leftarrow pos_i + 1$ ;  $pst_i \leftarrow pos_i$ 
19:      repeat  $Q_i \leftarrow qr_i$ ; broadcast REQ_W( $r, pst_i, est_i, Q_i$ )
20:      until  $Q_i \neq \perp$  and  $\forall p_j \in Q_i$  :
21:        RSP_W( $r, pst_i, p_j, \langle lre_j, pos_j, val_j \rangle$ ) received
22:         $rcv_i = \{ \langle lre_j, pos_j, est_j \rangle : p_j \in Q_i \wedge$ 
23:          RSP_W( $r, pst_i, p_j, \langle lre_j, pos_j, est_j \rangle$ ) received }
24:        if  $\exists \langle lre, -, - \rangle \in rcv_i : lre > lre_i$  then
25:          return( $\perp$ ) end if
26:         $pos_i \leftarrow \max\{pos \mid \langle r, pos, v \rangle \in rcv_i\}$ 
27:         $est_i \leftarrow \max\{v \mid \langle r, pos_i, v \rangle \in rcv_i\}$ 
28:      end while
29:    return( $est_i$ )
30: end function
31:
32: task: when REQ_R( $rd, Q$ ) is received:
33:   if  $p_i \in Q$  then
34:     if  $rd > lre_i$  then
35:        $pos_i \leftarrow g(pos_i, rd - lre_i)$ ;  $lre_i \leftarrow rd$  end if
36:       broadcast RSP_R( $rd, p_i, \langle lre_i, pos_i, est_i \rangle$ )
37:     end if
38:     broadcast REQ_R( $rd, Q$ )
39: end task
40:
41: task: when RSP_R( $rd, p_j, \langle lre_j, pos_j, est_j \rangle$ ) is received:
42:   broadcast RSP_R( $rd, p_j, \langle lre_j, pos_j, est_j \rangle$ )
43: end task
44:
45: task: when REQ_W( $rd, pos, est, Q$ ) is received:
46:   if  $p_i \in Q$  then
47:     if  $rd \geq lre_i$  then
48:        $pos_i \leftarrow g(pos_i, rd - lre_i)$ ;  $lre_i \leftarrow rd$ 
49:       if  $pos > pos_i$  then  $est_i \leftarrow est$ ;  $pos_i \leftarrow pos$ 
50:       else if  $pos = pos_i$  then
51:          $est_i \leftarrow \max\{est_i, est\}$  end if
52:     end if
53:     broadcast RSP_W( $rd, pos, p_i, \langle lre_i, pos_i, est_i \rangle$ )
54:   end if
55:   broadcast REQ_W( $rd, pos, est, Q$ )
56: end task
57:
58: task: when RSP_W( $rd, pos, p_j, \langle lre_j, pos_j, est_j \rangle$ ) is received:
59:   broadcast RSP_W( $rd, pos, p_j, \langle lre_j, pos_j, est_j \rangle$ )
60: end task

```

The original algorithm uses a selective multicast for both the read and write phases, i.e., messages are sent only to the processes in a quorum  $Q_i$ . Our algorithm uses broadcasts as defined in Section 2 (lines 8 and 19) and transmits  $Q_i$  with the message. All receiving processes will rebroadcast the message, but only the processes within  $Q_i$  will deliver it (lines 33 and 46).

**Theorem 3.** In our model augmented with  $\Pi\Sigma_{\perp,x}$ , Algorithm 3 ensures the properties of  $Alpha_x$ .

*Proof.* The modifications added to the original algorithms from [10] and [9] do not allow the algorithm to add new values, therefore the proof for validity in the original papers holds. Similarly, the proofs for obligation in [9] and quasi-agreement in [10] do not rely on any static connectivity assumption, and instead rely on algorithm behavioural properties which were not altered in our version. Therefore the original proofs hold for Algorithm 3.

Concerning termination, the only possibility for an invocation not to terminate is that process  $p_i$  waits forever for a response message in one of the repeat loops (lines 8–10 and 19–21). Let us assume by contradiction that  $p_i$  waits forever for responses. The liveness property of  $\Pi\Sigma_{\perp,x}$  ensures that eventually  $p_i$  only sends queries to correct processes and wait for responses from correct processes. Given that the set of correct processes is finite, the set of possible correct quorums is finite too. It follows that there is a correct quorum  $Q$  such that infinitely often,  $qr_i = Q$ , and therefore according to the quorum connectivity and self-inclusion properties of  $\Pi\Sigma_{\perp,x}$ , there are recurrent journeys between any process in  $Q$  and  $p_i$ . As a result, all the processes from  $Q$  will eventually receive the queries from  $p_i$ , and  $p_i$  will eventually receive the responses from the processes in  $Q$ , and therefore exit the repeat loop.  $\square$

### 6.3 $k$ -Set Agreement Algorithm

Given an  $Alpha_x$  object and a  $\Pi\Sigma_{\perp,x,y}$  failure detector, solving  $k$ -set agreement is simple. The algorithm given here is a straightforward adaptation of the one given in [9] for dynamic networks.

#### 6.3.1 $x$ -Set Agreement with $\Pi\Sigma_{\perp,x}$

Algorithm 4 solves  $x$ -set agreement in our model enriched with  $\Pi\Sigma_{\perp,x}$ .

In order to be well formed, an invocation of  $Alpha_x.propose(r, v)$  must guarantee that two processes cannot use the same round number  $r$ , and that successive round numbers for a given process are increasing. To this end, each process  $p_i$  initially computes  $prime_i$ , the  $i^{th}$  prime number.  $p_i$  then uses  $prime_i$  as its first round number, and multiplies it by  $prime_i$  after every round. As a result, the round number of  $p_i$  increases and is always a power of  $prime_i$ , which ensures that two distinct processes always use distinct round numbers.

**Algorithm 4.**  $x$ -Set agreement with  $Alpha_x$  using  $\Pi\Sigma_{\perp,x}$  in dynamic networks for process  $p_i$ .

```

1: init
2:    $dec_i \leftarrow \perp$ 
3:    $prime_i \leftarrow$  the  $i^{th}$  prime number
4:    $r_i \leftarrow prime_i$ 
5: end init
6:
7: function PROPOSE( $v_i$ )
8:   while  $dec_i = \perp$  do
9:     if  $leader_i = p_i$  then
10:       $dec_i \leftarrow Alpha_x.propose(r_i, v_i)$ 
11:       $r_i \leftarrow r_i \times prime_i$ 
12:     end if
13:   end while
14:    $decide(dec_i)$ 
15:   broadcast DECISION( $dec_i$ )
16: end function
17:
18: task: when DECISION( $d$ ) is received
19:   if  $dec_i = \perp$  then
20:      $dec_i \leftarrow d$ 
21:      $decide(d)$ 
22:   end if
23:   broadcast DECISION( $d$ )
24: end task

```

**Theorem 4.** In our model augmented with  $\Pi\Sigma_{\perp,x}$  and provided an  $Alpha_x$  object, Algorithm 4 solves the  $x$ -set agreement problem.

*Proof.* The test on line 8 ensures that the  $\perp$  value is never decided. From this point on, the validity of the  $Alpha_x$  object is enough to ensure the validity of  $x$ -set agreement. Similarly, the quasi-agreement property of  $Alpha_x$  is enough to ensure the agreement property of  $x$ -set agreement.

The eventual partial leadership property of  $\Pi\Sigma_{\perp,x}$  ensures that if every leader in  $L$  decides, then eventually every correct process will receive a DECISION message from a process in  $L$ . As a result, the proof for the termination property provided in [9] holds for Algorithm 4.  $\square$

### 6.3.2 $k$ -Set Agreement with $\Pi\Sigma_{\perp,x,y}$

As mentioned in [9], a simple  $k$ -set algorithm can be obtained by running  $y$  instances of Algorithm 4, the  $j^{th}$  one ( $1 \leq j \leq y$ ) relying on the component  $FD_i[j]$  of failure detector  $\Pi\Sigma_{\perp,x,y}$  for every process  $p_i$ . A process decides the value decided by the first of the  $y$  instances that terminates. As there are  $y$  instances of the algorithm and each of them can decide  $x$  values at most, it follows that at most  $xy$  values can be decided: therefore, the algorithm solves  $k$ -set agreement for  $k \geq xy$ .

## 7 RELATED WORK

In this section, we will first present a number of articles that offer solutions to agreement problems in dynamic systems, then we will compare the assumptions we use in this paper with existing models in the literature.

### 7.1 Agreement in Dynamic Systems

A number of papers have proposed solutions to agreement problems in dynamic networks, while relying on various timeliness, failure pattern, and connectivity assumptions.

The synchronous model is the most widely used to solve dynamic consensus in the literature. In [3], Kuhn et al. consider a model with a fixed set of processes communicating in synchronous rounds, and propose algorithms solving consensus, simultaneous consensus (all processes decide within the same round), and  $\Delta$ -coordinated consensus (all processes decide within  $\Delta$  rounds of each other). Biely et al. provide another algorithm for consensus in a similar model in [1], with weaker connectivity assumptions. In order to better formulate timeliness assumptions in the Time-Varying Graph formalism of [11], Gómez-Calzado et al. introduce in [19] the notion of timely journeys. They also propose an algorithm solving the Terminating Reliable Broadcast, which is equivalent to consensus in their synchronous model.

Fewer papers have studied asynchronous dynamic consensus. In [25], Taheri and Izadi propose a protocol solving the stronger problem of Byzantine consensus in an asynchronous dynamic system, using the necessary assumption that no more than  $\lfloor \frac{n-1}{3} \rfloor$  processes are faulty. Benchi et al. also provide in [5] an algorithm for asynchronous dynamic consensus under a similar failure pattern assumption.

From a failure detector perspective, some papers chose to implement the eventual leader detector  $\Omega$  [7] (the weakest failure detector to solve consensus in message passing environments with a majority of correct processes) in dynamic systems as a step towards consensus. Cao et al. in [12] study eventual leadership in dynamic systems, proposing a model in which the system is composed of two sets of nodes: fixed support stations forming a static complete graph with asynchronous communications, and mobile hosts communicating through the support stations. Eventual leader protocols for dynamic networks were also proposed by Gómez-Calzado et al. in [26] using partial synchrony assumptions, and by Arantes et al. in [27] using message pattern assumptions in a timer-free model.

To the best of our knowledge, only three papers have studied the problem of  $k$ -set agreement in dynamic systems. Biely et al. in [2] presented an algorithm for gracefully degrading consensus in synchronous dynamic networks. The algorithm solves consensus if the network conditions allow for it, and falls back on solving  $k$ -set agreement, otherwise. Another algorithm proposed by Sealfon and Sotiraki in [4] also relies on synchronous communications and on the assumption that every process knows an upper bound on the system membership.

### 7.2 Comparable Assumptions in the Literature

We attempt to put the strength of our assumptions into perspective by comparing them to some other existing models.

In [28], Afek and Gafni implement read and write operations in a dynamic synchronous message passing system. Although the underlying network is assumed to be complete, in each synchronous round a subset of edges will lose their messages. Therefore such a system can be modeled as a TVG where the edges that successfully deliver their message in a round are considered active in that round. As a result, the message adversary that decides which messages will go through can be compared to a connectivity assumption.

The paper defines the Traversal Path (TP) adversary as a model in which, for every synchronous round, the directed

graph defined by the successfully delivered messages in this round contains a directed path passing through all the nodes. This connectivity assumption is weaker than a TVG of class 5, because the traversal path is directed, which implies that every process can not necessarily communicate with every other. The comparison with class  $5-(\alpha, \gamma)$  is less straightforward. On the one hand, class  $5-(\alpha, \gamma)$  implies two-way connectivity whereas a TP adversary only requires one-way connectivity. On the other hand, class  $5(\alpha, \gamma)$  only requires connectivity between a limited number of nodes (as defined by the  $\alpha$  parameter) and allows network partitioning, whereas a TP adversary connects the entire system.

In [29], Biely et al. define and implement the generalized loneliness failure detector  $\mathcal{L}_k$  in a static and connected network. For this purpose they use the  $M^{anti(x)}$  message pattern model, which is defined by the  $x$ -Anti-Source. An  $x$ -Anti-Source is a process which is ensured to receive responses from  $x$  processes to every query it issues before it issues the next query.

This definition can be used in our model: if every process in the system is an  $x$ -Anti-Source for  $x \geq \lfloor \frac{n}{k+1} \rfloor + 1$ , then Assumption 2 (with  $\alpha = x$ ) and the quorum intersection property are ensured. However, the  $M^{anti(x)}$  model only requires  $x$  processes to be  $x$ -Anti-Sources. This would only be sufficient to implement quorums if  $x = n$ , since the intersection property must apply to every process in the system.

In [27], Arantes et al. implement the  $\Omega$  failure detector in an asynchronous TVG of class 5. For this purpose they define the Stable Responsiveness Property ( $SRP$ ). A correct process  $p$  satisfies the  $SRP$  at time  $t$  if and only if, after  $t$ , all nodes in  $p$ 's neighborhood receive a response from  $p$  to every one of their queries within the first  $\alpha$  responses.

The definition of the  $SRP$  can be compared to the definition of an eventually winning process. Both properties enable a leader election mechanism by assuming that after some time, some process is among the first to respond to the queries of its neighbors. However,  $SRP$  applies to every process that shares a link with the leader after  $t$ , even for a moment, whereas the property of an eventually winning process  $p_i$  only applies to the processes of  $R_i$ , meaning those processes that interact infinitely often with  $p_i$ . This means that a process can join the neighborhood of an eventually winning leader and leave it later on, which is not possible with a process satisfying the  $SRP$ . But while the properties of an eventually winning leader are required to apply to a smaller subset of processes, those properties are stronger. Process  $p_i$  is not only required to respond to every query from its neighbors in time, it must also be the fastest to respond. Additionally, the processes within  $R_i$  are expected to communicate with each other to some extent, which is not necessary to satisfy the  $SRP$ .

## 8 CONCLUSION

In this paper we adapted the existing  $\Pi\Sigma_{x,y}$  failure detector to unknown dynamic systems by using the  $\perp$  default value to deal with missing information and by adding connectivity properties to the failure detector definition. We obtained the  $\Pi\Sigma_{\perp,x,y}$  failure detector, which is sufficient to solve  $k$ -set agreement in unknown dynamic systems with  $k \geq xy$ .

We then provided an algorithm implementing  $\Pi\Sigma_{\perp,x,y}$  in a Time-Varying Graph of class  $5-(\alpha, \gamma)$ , along with the connectivity and message pattern assumptions it relies on.

Finally, we adapted an existing algorithm to solve  $k$ -set agreement in unknown dynamic networks augmented with  $\Pi\Sigma_{\perp,x,y}$  ( $k \geq xy$ ).

Future research could attempt to further weaken the system model by removing the assumption of synchronous processes. The connectivity model would then need to be adapted, since the synchrony of processes allows the algorithm to take advantage of the time windows provided by  $\gamma$ -journeys. Such a change would be a challenge, because the other approaches used to ensure reachability in a TVG ([19]) rely on point to point communications, which is not applicable in an unknown network.

Another research direction would be to solve other problems in similarly weak models, such as the implementation of shared registers in a unknown dynamic message passing system.

## REFERENCES

- [1] M. Biely, P. Robinson, and U. Schmid, "Agreement in Directed Dynamic Networks," in *SIROCCO*, vol. 7355, 2012, pp. 73–84.
- [2] M. Biely, P. Robinson, U. Schmid, M. Schwarz, and K. Winkler, "Gracefully Degrading Consensus and  $k$ -Set Agreement in Directed Dynamic Networks," *CoRR*, vol. abs/1501.02716, 2015.
- [3] F. Kuhn, Y. Moses, and R. Oshman, "Coordinated consensus in dynamic networks," in *PODC*, 2011, pp. 1–10.
- [4] A. Sealon and A. Sotiraki, "Agreement in Partitioned Dynamic Networks," *CoRR arXiv:1408.0574*, 2014.
- [5] A. Benchi, P. Launay, and F. Guidec, "Solving consensus in opportunistic networks," in *ICDCN*, 2015, pp. 1:1–1:10.
- [6] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *JACM*, vol. 43, no. 2, pp. 225–267, 1996.
- [7] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *JACM*, vol. 43, no. 4, pp. 685–722, 1996.
- [8] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *JACM*, vol. 32, no. 2, pp. 374–382, 1985.
- [9] A. Mostéfaoui, M. Raynal, and J. Stainer, "Chasing the Weakest Failure Detector for  $k$ -Set Agreement in Message-Passing Systems," in *NCA 2012*, 2012, pp. 44–51.
- [10] Z. Bouzid and C. Travers, "(anti- $\Omega^x \times \Sigma_z$ )-Based  $k$ -Set Agreement Algorithms," in *OPDIS 2010*, vol. 6490, 2010, pp. 189–204.
- [11] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *IJPEDES*, vol. 27, no. 5, pp. 387–408, 2012.
- [12] J. Cao, M. Raynal, C. Travers, and W. Wu, "The eventual leadership in dynamic mobile networking environments," in *PRDC*, 2007, pp. 123–130.
- [13] F. Kuhn, N. A. Lynch, and R. Oshman, "Distributed computation in dynamic networks," in *STOC 2010*. ACM, 2010, pp. 513–522.
- [14] T. Rieutord, L. Arantes, and P. Sens, "Détecteur de défaillances minimal pour le consensus adapté aux réseaux inconnus," in *Algotel*, 2015.
- [15] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui, "Tight failure detection bounds on atomic object implementations," *JACM*, vol. 57, no. 4, 2010.
- [16] F. Bonnet and M. Raynal, "Looking for the Weakest Failure Detector for  $k$ -Set Agreement in Message-Passing Systems: Is  $\Pi_k$  the End of the Road?" in *SSS 2009*, vol. 5873, 2009, pp. 149–164.
- [17] P. Zielinski, "Anti-Omega: the weakest failure detector for set agreement," in *PODC 2008*, 2008, pp. 55–64.
- [18] M. Biely, P. Robinson, and U. Schmid, "Easy impossibility proofs for  $k$ -set agreement in message passing systems," in *OPDIS 2011*, 2011, pp. 299–312.
- [19] C. Gómez-Calzado, A. Casteigts, A. Lafuente, and M. Larrea, "A Connectivity Model for Agreement in Dynamic Systems," in *EuroPar*, 2015.



- [20] A. Mostéfaoui, E. Mourgaya, and M. Raynal, "Asynchronous implementation of failure detectors," in *DSN*, 2003, pp. 351–360.
- [21] I. F. Akyildiz, X. Wang, and W. Wang, "Wireless mesh networks: a survey," *Computer Networks*, vol. 47, no. 4, pp. 445–487, 2005.
- [22] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.
- [23] R. Guerraoui and M. Raynal, "The Alpha of Indulgent Consensus," *Comput. J.*, vol. 50, no. 1, pp. 53–67, 2007. [Online]. Available: <http://dx.doi.org/10.1093/comjnl/bxl046>
- [24] M. Raynal and C. Travers, "In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-Free Set Agreement," in *OPODIS 2006*, vol. 4305, pp. 3–19. [Online]. Available: [http://dx.doi.org/10.1007/11945529\\_2](http://dx.doi.org/10.1007/11945529_2)
- [25] E. Taheri and M. Izadi, "Byzantine consensus for unknown dynamic networks," *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1587–1603, 2015.
- [26] C. Gómez-Calzado, A. Lafuente, M. Larrea, and M. Raynal, "Fault-Tolerant Leader Election in Mobile Dynamic Distributed Systems," in *PRDC*, 2013, pp. 78–87.
- [27] L. Arantes, F. Greve, P. Sens, and V. Simon, "Eventual leader election in evolving mobile networks," in *OPODIS*, vol. 8304, 2013, pp. 23–37.
- [28] Y. Afek and E. Gafni, "Asynchrony from synchrony," in *ICDCN 2013*, 2013, pp. 225–239.
- [29] M. Biely, P. Robinson, and U. Schmid, "The generalized loneliness detector and weak system models for k-set agreement," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 4, pp. 1078–1088, 2014.