



**HAL**  
open science

## A Failure Detector for k-Set Agreement in Asynchronous Dynamic Systems

Élise Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

► **To cite this version:**

Élise Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens. A Failure Detector for k-Set Agreement in Asynchronous Dynamic Systems. [Research Report] RR-8727, UPMC Sorbonne Universités/CNRS/Inria - EPI REGAL; INRIA. 2015. hal-01151739v2

**HAL Id: hal-01151739**

**<https://inria.hal.science/hal-01151739v2>**

Submitted on 2 Jul 2015

**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.



# A Failure Detector for $k$ -Set Agreement in Dynamic Systems

Denis Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

**RESEARCH  
REPORT**

**N° 8727**

May 2015

Project-Team Regal





## A Failure Detector for $k$ -Set Agreement in Dynamic Systems

Denis Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

Project-Team Regal

Research Report n° 8727 — version 2 — initial version May 2015 —  
revised version Juillet 2015 — 19 pages

**Abstract:** The  $k$ -set agreement problem is a generalization of the consensus problem where processes can decide up to  $k$  different values. Very few papers have tackled this problem in dynamic networks, and to the best of our knowledge, every algorithm proposed so far for  $k$ -set agreement in dynamic networks assumed synchronous communications or made strong failure pattern assumptions. Exploiting the formalism of the Time-Varying Graph model, this paper proposes a new quorum-based failure detector for solving  $k$ -set agreement in dynamic networks with asynchronous communications. We present two algorithms that implement this new failure detector using graph connectivity and message pattern assumptions. We also provide an algorithm for solving  $k$ -set agreement using our new failure detector.

**Key-words:**  $k$ -set agreement, failure detectors, dynamic networks, time-varying graphs, asynchronous communications, evolving membership.

**RESEARCH CENTRE  
PARIS – ROCQUENCOURT**

Domaine de Voluceau, - Rocquencourt  
B.P. 105 - 78153 Le Chesnay Cedex

# Un Détecteur de Défaillances pour le $k$ -Accord dans les Systèmes Dynamiques

**Résumé :** Le problème du  $k$ -accord est une généralisation du problème du consensus dans lequel les processus peuvent décider jusqu'à  $k$  différentes valeurs. Très peu de papiers ont étudié ce problème dans des réseaux dynamiques, et tous les algorithmes de  $k$ -accord dynamique dont nous avons connaissance font l'hypothèse de communications synchrones ou d'un nombre de fautes limité. En utilisant le formalisme du Time-Varying Graph, ce papier propose un nouveau détecteur de défaillances résolvant le  $k$ -accord dans les réseaux dynamiques avec des communications asynchrones. Nous présentons deux algorithmes qui implémentent ce nouveau détecteur de défaillances, sous réserve d'hypothèses à motif de messages et sur la connectivité du graphe. Nous fournissons également un algorithme résolvant le  $k$ -accord en utilisant notre nouveau détecteur.

**Mots-clés :**  $k$ -accord, détecteurs de défaillances, réseaux dynamiques, time-varying graphs, communications asynchrones.

## 1 Introduction

Modern distributed architectures such as clouds or ad-hoc networks are characterized by their high number of nodes and strong dynamics. On the other hand, traditional models and algorithms of distributed computing are not always adapted to face the challenges brought by those new architectures. Indeed, they rely on static and known membership networks, where the communication graph does not change for the whole duration of the run and every process knows the number of processes in the system and each of their unique identifiers. By contrast, in dynamic networks, nodes can join and leave the system during the run and communication between them varies over the time.

Agreement problems such as *consensus* are the keystone of distributed computing problems and have numerous applications. Notably, machine state replication in a distributed system requires to solve an agreement problem. However, these problems have been less studied in dynamic models than they have been in traditional ones. In this paper, we are interested in a particular agreement problem, namely *k-set agreement* [13], and how to solve it in dynamic systems with asynchronous communications. The *k-set agreement* problem is a generalization of the *consensus* problem where processes eventually agree on at most  $k$  proposed different values. However, it cannot be solved in asynchronous systems prone to  $f$  failures when  $k \leq f$ . In order to circumvent such an impossibility, solutions enriched with failure detectors have been exploited. Failure detectors [12] are distributed oracles that provide processes with information, not always correct, on processes failures. They address questions such as the minimal information about failures needed to implement agreement problems.

In asynchronous message passing model with  $n$  nodes and  $f < n$ , it has been proved in [15] that the weakest failure detector for solving *consensus* (i.e., *1-set agreement*) is the class of eventual leader failure detector, denoted  $\Omega$ , combined with the quorum failure detector, denoted  $\Sigma$  [14]. For *k-set agreement*, the weakest failure detector in message passing systems is unknown but the failure detector  $\Sigma_k$  was proved to be necessary [6].

Recently, Rieutord et al. [24] proposed the failure detector  $\Sigma_{\perp}$ , which is an adaptation of  $\Sigma$  [14] in the absence of initial information regarding system membership. It is, therefore, suitable for implementations in dynamic networks. Hence, our proposal in this paper is to combine the properties of both  $\Sigma_k$  and  $\Sigma_{\perp}$  in order to solve *k-set agreement* in dynamic systems.

For modeling the dynamics of the system and evolving communication between nodes, we exploit the formalism of the *Time-Varying Graphs (TVG)*, proposed by Casteigts et al. in [10]. In this article, the authors defined several classes of *TVGs* and compare them according to the strength of the assumptions made on graph connectivity. Thus, by introducing new classes, which are extensions of *TVG* class 5 (*recurrent connectivity*), we can express our required model assumptions.

**Contributions.** This paper brings four main contributions:

1. The conception of new *TVG* classes for efficiently implementing *k-set agreement* in dynamic systems with asynchronous communications.
2. The new failure detector  $\Sigma_{\perp,k}$ , which combines the properties of both  $\Sigma_k$  and  $\Sigma_{\perp}$ .
3. Two algorithms for the failure detector  $\Sigma_{\perp,k}$ , making different connectivity and message pattern assumptions.
4. An algorithm for solving *k-set agreement* in our model, using  $\Sigma_{\perp,z}$  with  $k \geq n - \lfloor \frac{n}{z+1} \rfloor$ . The algorithm is adapted from a protocol proposed in [7], solving *k-set agreement* in static networks with known membership.

Our system model is dynamic in different aspects: firstly, processes can join or leave the system during the run. Secondly, network links can be activated or deactivated during the run, which means that the communication graph evolves over time.

**Roadmap.** The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 presents background on models and the formal definition of the *k-set agreement* problem. Section 4 introduces the failure detector class  $\Sigma_{\perp,k}$  and Section 5 presents the new *TVG* classes used by our  $\Sigma_{\perp,k}$  algorithms whose implementations are presented in Section 6. Section 7 proposes an algorithm solving *k-set agreement* using  $\Sigma_{\perp,k}$  and Section 8 concludes the paper.

## 2 Related Work

**Dynamic system models.** In order to solve problems in modern dynamic networks, a number of papers in the literature have introduced formalisms to express dynamic system models.

In [1], Aguilera presents different system models where an infinite number of processes can join the system during a given run.

Cao et al. presented in [8] a model made of two sets of nodes: fixed support stations forming a static complete graph, asynchronous but entirely known, and mobile hosts which follow the finite arrival model of [1] and communicate through the support stations.

Ferreira introduced in [17] a model, namely *Evolving Graphs*, in which the communication graph evolves over time. Based also on communication graph evolution, Casteigts et al. presented in [10] the formalism of the *Time-Varying Graph (TVG)* and defined several classes of TVGs, according to different graph connectivity assumptions. As our work uses the TVG model, its formalism is described in Section 3.

Kuhn et al. proposed in [20] a model considering an evolving communication graph where connectivity assumptions rely on stable subgraphs connected for a certain number of synchronous rounds. Biely et al. presented in [4] a dynamic model, which is close to the model of Kuhn et al. [20], but based on directed graphs and weaker connectivity assumptions.

A survey of dynamic system models can be found in [22] and [19].

**Algorithms solving *consensus* in dynamic systems** Some algorithms have been presented to solve the *consensus* problem in dynamic systems. Those solutions make strong assumptions on either the timeliness of communications or the number of process failures.

An algorithm for *consensus* exploiting the model of Kuhn et al. [20] is provided in [21], along with an algorithm for *simultaneous consensus* (all the nodes decide in the same synchronous round) and another one for  *$\Delta$ -coordinated consensus* (all the nodes decide within  $\Delta$  rounds of each other).

An algorithm for *consensus* is also presented in [4], considering assumptions on *vertex-stable root component*: in every synchronous round, there must be exactly one strongly connected component that has only out-going links to some of the remaining processes and can reach every process in the system.

Recently, an algorithm for *consensus* in asynchronous dynamic systems has been presented in [3]. The solution makes use of the One-Third Rule algorithm and epidemic routing. It assumes that no more than a third of the processes involved in the consensus crash.

Several implementations of the eventual leader failure detector  $\Omega$  [11], which is necessary to solve *consensus*, have been adapted for dynamic systems in [8, 2, 19]. The algorithm in [19] relies on partial synchrony assumptions, whereas the ones in [8, 2] are asynchronous and use message pattern assumptions.

**Algorithms solving  $k$ -set agreement in dynamic systems** Very few articles have attempted to solve the  $k$ -set agreement problem in dynamic networks.

Exploiting the formalism of [20] and considering an upper bound on the number of processes, Sealfon and Sotiraki present in [25] an algorithm for solving  $k$ -set agreement in *partitioned dynamic networks*.

Biely et al. in [5] proposed in their model [4] an algorithm for *consensus* which gracefully degrades to  $k$ -set agreement whenever network conditions do not guarantee *consensus*. We should point out that both algorithms assume synchronous communications between processes: to the best of our knowledge, there is no algorithm in the literature that solves the  $k$ -set agreement problem in dynamic networks with asynchronous communications.

### 3 Models and Definitions

#### 3.1 Process Model

The system is made up of a finite set of  $n$  processes denoted  $\Pi = \{p_1, p_2, \dots, p_n\}$ . Processes in the system are uniquely identified, but are initially only aware of their own identity. We assume that there is a bound on the relative speed of processes: the latter are, therefore, synchronous.

Assuming that processes are synchronous may appear as a strong assumption, but it is very weak compared to synchronous communication. Indeed, if communications are not bounded, to the other processes point of view the synchrony of processes cannot be used for inter-process synchronization. To the opposite, if communications are synchronous and processes asynchronous, a message sent to all neighbors will arrive to all in a fixed range of time and may be used to synchronize their local estimation of time. This difference between the computation synchrony and communication synchrony power is used in Dwork et al. paper on partial synchrony [16].

Processes follow the *n-arrival model* of [1]: processes may join and leave the system, but only a total of  $n$  processes can join the system in a run. A run is a sequence of steps executed by the processes while respecting the causality of operations (each received message has been previously sent). A process may leave the system and rejoin it later, but then a new identity will be attributed to it: for all intents and purposes, it will be treated as a new process.

A process that never leaves the system in a run is said to be *correct* in that run, otherwise it is *faulty*: we make no difference between a process that crashes and a process that purposely leaves the system. The set of all *correct* processes in a run is denoted  $\mathcal{C}$ . We assume an upper bound  $f < n$  on the number of *faulty* processes in the system.

#### 3.2 Communication Model

**Time-Varying Graph.** We model the system dynamics using the formalism of the *Time-Varying Graph (TVG)*, as introduced in [10]. The topology of the network is dynamic, which means that the relations between two nodes take place over a time span  $\mathcal{T} = \mathbb{N}$ .

**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  $t \in \mathcal{T}$ .*

Note that the edges in  $E$  can be directed or not.

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

Although we make use of the latency function  $\zeta$  to express communication constraints, processes do not have access to  $\zeta$  and its values are expected to be finite but not necessarily bounded: communications in the system are thus *asynchronous*.

**Journeys.** The connectivity assumptions of our model are expressed in terms of *journeys*, as defined using the formalism of the *TVG* model.

**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 denoted  $\text{departure}(\mathcal{J})$  and  $t_m + \zeta(e_m, t_m)$  is denoted  $\text{arrival}(\mathcal{J})$ . Intuitively, a *journey* is a path over time. As such, it has both a topological length ( $|\mathcal{J}| = m$ , the length of the path in  $G$ ) and a temporal length ( $\text{arrival}(\mathcal{J}) - \text{departure}(\mathcal{J})$ ).

If a *journey* exists between two nodes  $u$  and  $v$ , we say that  $u$  can *reach*  $v$ , which is denoted  $u \rightsquigarrow v$ . We denote  $\mathcal{J}_{(u,v)}^*$  all the journeys starting at node  $u$  and ending at node  $v$ .

**Definition 3** (Direct journey). *A direct journey  $\mathcal{J}$  is a journey such that every next edge in  $\mathcal{J}$  is directly available:*

$$\forall i, 1 \leq i < |\mathcal{J}| : \rho(e_{i+1}, t_i + \zeta(e_i, t_i)) = 1 .$$

**Broadcast primitive.** Processes communicate by using a *broadcast* primitive. When a process  $p_i$  initiates a *broadcast* of message  $m$ , it simply sends  $m$  to every process currently in its direct neighborhood, including itself. This is therefore a low-level *broadcast*, which notably does not handle message forwarding (forwarding will be explicitly handled by our algorithms).

**Channels.** Channels are *fair-lossy*, which means that messages may be lost, but, if a process  $p_i$  broadcasts a message  $m$  infinitely often such that, infinitely often, the edge connecting  $p_i$  and a *correct* process  $p_j$  is active at the time of one of these broadcasts and for the entire transfer time, then  $p_j$  will receive  $m$  an infinite number of times. There is no creation or alteration of messages, but messages may be duplicated (provided that each message is only duplicated a finite number of times) or arrive in any order (in particular, we do not require channels to be FIFO).

**TVG classes.** Several classes of *TVGs* have been defined in [10] with respect to temporal properties on the network dynamics. They imply necessary conditions and impossibility results for distributed computing and algorithms. We are particularly interested in class 5:

**Definition 4** (Class 5: recurrent connectivity). *TVGs of class 5 ensure that every process can reach every other process infinitely often:*

$$\forall u, v \in V, \forall t \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}_{(u,v)}^* : \text{departure}(\mathcal{J}) > t .$$

### 3.3 The $k$ -Set Agreement Problem

The problem of *k-set agreement* was introduced by Chaudhuri in [13] as a generalization of the *consensus* problem for  $1 \leq k \leq n - 1$ . Each process starts by proposing a value, and each correct process eventually decides a value in such a way that the following three properties are satisfied:

- **Validity.** A decided value is a proposed value.
- **Termination.** Every correct process eventually decides a value.
- **Agreement.** At most  $k$  different values are decided.

## 4 A New Failure Detector: $\Sigma_{\perp,k}$

The *quorum failure detector*  $\Sigma$  [14] provides every process with a set of process identities, denoted *quorum*, such that any two *quorums* formed at any time necessarily intersect, and eventually all *quorums* are included in  $\mathcal{C}$ . Note that this property must hold over time: two *quorums* formed at two different times by two different processes must intersect.

Similarly, the *quorum failure detector family*  $\Sigma_k$  [6] provides processes with *quorums* of process identities such that two out of any  $k + 1$  *quorums* necessarily intersect, and eventually all *quorums* are included in  $\mathcal{C}$ .

Given that  $\Sigma_k$  was proved in [6] to be necessary for solving *k-set agreement* in asynchronous systems, it seems natural to attempt to implement it using the *TVG* formalism as a stepping stone towards dynamic *k-set agreement*. However, the intersection property of  $\Sigma_k$  requires to be fulfilled from the start of the run, which poses a problem in a dynamic system where processes do not necessarily know each other's identities from the start.

A solution to this problem for the case of  $\Sigma$  ( $k = 1$ ) was proposed in [24] by the  $\Sigma_{\perp}$  failure detector, which outputs either the normal output of  $\Sigma$  or the special value  $\perp$ . A process can return  $\perp$  instead of a *quorum* to indicate that it has not yet gathered sufficient knowledge of the system to form a *quorum*. We apply the same concept to  $\Sigma_k$  in order to propose a failure detector which can be used to solve *k-set agreement* in dynamic asynchronous systems.

The  $\Sigma_{\perp,k}$  detector provides each process  $p$  with a set of process identities (denoted  $\Sigma_{\perp,k,p}(t)$  at time  $t$ ) such that the following properties are satisfied:

**Property 1** (Intersection). *Out of any  $k + 1$  non- $\perp$  quorums, at least two intersect:  $\forall t_1, \dots, t_{k+1} \in \mathcal{T}, \forall p_1, \dots, p_{k+1} \in \Pi, \exists i, j :$*

$$\begin{aligned} 1 \leq i \neq j \leq k + 1, (\Sigma_{\perp,k,p_i}(t_i) \neq \perp \wedge \Sigma_{\perp,k,p_j}(t_j) \neq \perp) \\ \Rightarrow \Sigma_{\perp,k,p_i}(t_i) \cap \Sigma_{\perp,k,p_j}(t_j) \neq \emptyset . \end{aligned}$$

**Property 2** (Completeness). *Eventually, all quorums are different from  $\perp$  and contain only correct processes:  $\exists \tau \in \mathcal{T}, \forall p \in \mathcal{C}, \forall t \in \mathcal{T}, t \geq \tau :$*

$$\Sigma_{\perp,k,p}(t) \neq \perp \wedge \Sigma_{\perp,k,p}(t) \subseteq \mathcal{C} .$$

Intuitively,  $\Sigma_{\perp,k}$  ensures the same properties as  $\Sigma_k$  except that initially, the detector can return the special value  $\perp$  instead of a *quorum* to indicate that sufficient knowledge about process identities has not been reached, and therefore the intersection property is not expected to hold yet. By convention, a crashed process returns  $\perp$  forever.

For the sake of simplicity, we assume that processes always include themselves in their respective non- $\perp$  *quorums*. Our algorithms in Section 6 will implement such an inclusion.

## 5 New *TVG* Classes for reliable message transmission

As described in Section 3, *TVGs* of class 5 (*recurrent connectivity*) ensure that every process can *reach* every other process infinitely often. However, the latter is insufficient to guarantee that a

message will eventually cross an edge, even if the message is sent infinitely often: the edge could be present only in between two emissions of the message. A naive solution would be to send the message constantly, such that at any instant there is a message being sent. This assumption is unrealistic, as it would require processes to send an infinite number of messages in a finite time.

A solution to this problem, proposed in [9], is to assume that when an edge appears or disappears, the adjacent nodes are notified of the corresponding event without delay. With this information, it is possible to implement a *send\_retry* primitive which re-sends the message upon reappearance of the edge. Eventual message reception is thus ensured, provided that there is a long enough edge activation period in the future.

In [18], Gómez-Calzado et al. consider that the assumption of instantaneous edge detection is too strong. Instead, they define  $\beta$ -*journeys*, in which edges are guaranteed to stay active at least  $\beta$  time, with  $\beta$  strictly longer than the message transmission time. However, the authors consider *synchronous* communications and therefore define  $\beta$ -*journeys* using an upper bound on the transmission time. To overcome these drawbacks, we define  $\gamma$ -*journeys*.

Similarly to  $\beta$ -*journeys*,  $\gamma$ -*journeys* assume that every edge stays active strictly longer than the transfer delay in order to give the sender process the necessary time to send the message.

**Definition 5** ( $\gamma$ -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  $t_i + \max_{0 \leq j \leq \gamma} \{j + \zeta(e_i, t_i + j)\}$ .
- $\forall i, 1 \leq i < |\mathcal{J}|, t_{i+1} \geq t_i + \max_{0 \leq j \leq \gamma} \{j + \zeta(e_i, t_i + j)\}$  .

We then define *direct  $\gamma$ -journeys* similarly to *direct journeys*. We call  $\mathcal{J}_{(u,v)}^\gamma$  the set of all  $\gamma$ -*journeys* from  $u$  to  $v$ , and  $\mathcal{J}_{(u,v)}^{d\gamma}$  the set of all *direct  $\gamma$ -journeys* from  $u$  to  $v$ . We then obtain the following new class of *TVGs*:

**Definition 6** (Class 5- $\gamma$ :  $\gamma$ -recurrent connectivity). *TVGs of class 5- $\gamma$  ensure that every process can reach every other process infinitely often through  $\gamma$ -journeys:*

$$\forall u, v \in V, \forall t \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}_{(u,v)}^\gamma : \text{departure}(\mathcal{J}) > t .$$

Class 5- $\gamma'$  ( $\gamma$ -*direct recurrent connectivity*) is defined similarly, except that it uses  $\mathcal{J}_{(u,v)}^{d\gamma}$  instead of  $\mathcal{J}_{(u,v)}^\gamma$ .

Class 5- $\gamma$  is strictly stronger than class 5. Trivially,  $\forall \gamma_1, \gamma_2 : \gamma_1 \geq \gamma_2 \Rightarrow \text{class } 5-\gamma_1 \subseteq \text{class } 5-\gamma_2$  (class 5 would be equivalent to “class 5-0”, if  $\gamma$  could be equal to 0 in Definition 5).

Class 5- $\gamma'$  is significantly stronger than class 5- $\gamma$ , as the assumption of recurrent *direct journeys* considerably reduces the dynamics of the system.

Although classes 5- $\gamma$  and 5- $\gamma'$  present the advantage of being easily comparable to class 5, they are slightly too strong for our algorithms: we only require a limited number of nodes to be connected. To this end, we define two more *TVG* classes:

**Definition 7** (Class 5- $(\alpha, \gamma)$ :  $(\alpha, \gamma)$ -recurrent connectivity). *Every process can reach and be reached through  $\gamma$ -journeys infinitely often by at least  $\alpha$  correct processes, including itself.*

$$\begin{aligned} \forall p_i \in \mathcal{C}, \exists P_i \subseteq \mathcal{C}, |P_i| \geq \alpha, \forall t \in \mathcal{T}, \forall p_j \in P_i, \\ \exists \mathcal{J}_i \in \mathcal{J}_{(p_i, p_j)}^\gamma : \text{departure}(\mathcal{J}_i) \geq t \wedge \\ \exists \mathcal{J}_j \in \mathcal{J}_{(p_j, p_i)}^\gamma : \text{departure}(\mathcal{J}_j) \geq t . \end{aligned}$$

**Definition 8** (Class  $5-(\alpha, \gamma)'$ :  $(\alpha, \gamma)$ -direct recurrent receiver connectivity). *Every process can be reached through direct  $\gamma$ -journeys infinitely often by at least  $\alpha$  correct processes, including itself.*

$$\forall p_i \in \mathcal{C}, \exists P_i \subseteq \mathcal{C}, |P_i| \geq \alpha, \forall t \in \mathcal{T}, \forall p_j \in P_i, \\ \exists \mathcal{J} \in \mathcal{J}_{(p_j, p_i)}^{d\gamma} : \text{departure}(\mathcal{J}) \geq t .$$

Note that class  $5-(\alpha, \gamma)$  requires two-way *recurrent connectivity*, whereas class  $5-(\alpha, \gamma)'$  does not. The latter, however, requires *direct  $\gamma$ -journeys*.

## 6 Algorithms for $\Sigma_{\perp, k}$ in Dynamic Networks

In this section, we present two implementations of  $\Sigma_{\perp, k}$  using the *TVG* classes presented in Section 5, relying on message pattern assumptions. The first algorithm requires a *TVG* of class  $5-(\alpha, \gamma)'$  whereas the second requires class  $5-(\alpha, \gamma)$ .

The principle of both algorithms is the following: each process broadcasts their identity regularly and, infinitely often, waits for messages from  $\alpha$  processes and forms a new quorum with the received process identities. Since our broadcast primitive does not handle forwarding, each process needs to rebroadcast the information it receives. The properties of class  $5-(\alpha, \gamma)$  and class  $5-(\alpha, \gamma)'$  ensure that *correct* processes will always receive enough messages to form quorums.

The algorithms are compatible with any value of  $\alpha$  that fits the definition of our *TVG* classes, but the processes are required to know the same  $\alpha$  value ( $\alpha \leq |\mathcal{C}|$ ).  $\gamma$  is the maximum delay between two broadcasts of the same message by a process. We also require processes to receive their own broadcast messages within  $\gamma$  units of time. Note that the bound value of  $\gamma$  implies that the relative speed of processes must be bounded: processes are therefore *synchronous*.

Since we make use of infinite rebroadcasts, messages originally issued by *faulty* processes will be continuously retransmitted by the other correct processes, even after all *faulty* processes have left the system preventing the satisfaction of the *completeness* property. As both algorithms use different solutions to this problem, it will be discussed further in the subsections presenting the algorithms themselves.

### 6.1 Message pattern

In order to ensure the *intersection* property, some assumptions must be added. Instead of using a constraint on the ratio of *correct* processes, we present here assumptions based on the message pattern approach proposed by Mostéfaoui et al. [23].

The message pattern model consists in assuming some properties on the relative order of message deliveries. When a certain number of messages ( $\alpha$ ) are waited by processes for continuing algorithm progress, the principle is to assume that the message from some specific process will be among the first received. This model provides asynchronous implementations and avoid timer-based approaches.

We propose two different message pattern assumptions, either of which is sufficient to implement  $\Sigma_{\perp, k}$ . Both are generalizations of the message pattern assumption used by Rieutord et al. in [24] applied to *k-set agreement*.

**Assumption 1** (Multiple winning quorums).

$\exists Q_{w1}, Q_{w2}, \dots, Q_{wk} \subseteq \Pi : \forall p \in \Pi$ , every time  $p$  attempts to form a new quorum,  $\exists i \in [1, k]$  such that  $Q_{wi} \neq \emptyset$  and out of the next  $\alpha$  processes from which  $p$  receives a message, at least  $\lfloor \frac{|Q_{wi}|}{2} \rfloor + 1$  of them are in  $Q_{wi}$ .

We refer to the  $Q_{wi}$  sets as *winning quorums*. The property on the size of each *winning quorum* ensures that any two processes forming a *quorum* using the same *winning quorum* will necessarily intersect. Since there are only  $k$  different such *winning quorums*, two out of any  $k + 1$  *quorums* formed by the algorithm will intersect. Assumption 1 implies that there is at least one *winning quorum*  $Q_{wi}$  such that a strict majority of the processes in  $Q_{wi}$  are *correct*.

**Assumption 2** (Global winning quorum).  $\exists Q_w \subseteq \Pi, Q_w \neq \emptyset : \forall p \in \Pi$ , every time  $p$  attempts to form a new quorum, out of the next  $\alpha$  processes from which  $p$  receives a message, at least  $\lfloor \frac{|Q_w|}{k+1} \rfloor + 1$  of them are in  $Q_w$ .

This assumption implies that at least  $\lfloor \frac{|Q_w|}{k+1} \rfloor + 1$  processes in  $Q_w$  are *correct*. Assumption 2 requires a single *winning quorum*, but the constraint on the size of  $Q_w$  is weaker than the one of Assumption 1. It can be shown that these two assumptions are not equivalent.

**Theorem 1.** *Assumption 1 and Assumption 2 are incomparable and none is weaker than the other one.*

*Proof.* **Assumption 2  $\not\Rightarrow$  Assumption 1 :** Consider  $k = 2$  and  $\alpha = 2$ . The single *winning quorum* provided by Assumption 2,  $Q_w$ , can contain at most 5 processes. Consider that for such a *winning quorum* of size 5, two *winning quorums* can be found satisfying Assumption 1. Such *quorums* must contain at most 3 elements with 2 responses required, or of size 1 and 1 response. Among the possible couple of responses of processes from  $Q_w$ , the case of *winning quorums* of size 3 can cover for at most 3 possible couples satisfying the condition of Assumption 1 and the one of size 1 can cover for at most 4. So Assumption 1 can cover for at most 8 possible selection of couple of responses from  $Q_w$ , but there is 10 possible combinations for selecting 2 elements in a set of 5.

**Assumption 1  $\not\Rightarrow$  Assumption 2 :** Let us consider  $\alpha = 2$  and that all of Assumption 1 *winning quorums* are disjoint and composed of three processes. Due to size considerations, if Assumption 2 is verified, the associated *winning quorum*  $Q_w$  must be composed of at most  $2k + 1$  processes and thus cannot contain all  $3k$  processes of Assumption 1 *winning quorums*. Therefore, a quorum formed with only one of  $Q_w$  process may be formed respecting Assumption 1, implying that  $|Q_w| \leq k$ . Furthermore, this imply that a  $Q_{wi}$  with at most one process in  $Q_w$  exists and so a valid quorum can be formed with the two processes of  $Q_{wi}$  that are not in  $Q_w$ . In such a scenario no *winning quorum* respecting Assumption 2 can be found.  $\square$

Note that if  $\alpha \geq \lfloor \frac{n}{k+1} \rfloor + 1$ , then Assumption 2 is trivially verified with  $Q_w = \Pi$ . This particular case is the method used to implement  $\Sigma_k$  in static networks in [7].

The two algorithms that we present in the following require either Assumption 1 or Assumption 2 to hold.

## 6.2 A Message Expiration-Based Algorithm for $\Sigma_{\perp, k}$

Algorithm 1 provides *completeness* by limiting the number of times a message can be rebroadcast. It requires a *TVG* of class  $5-(\alpha, \gamma)'$ .

**Notations.** Broadcast messages contain the parameters: (1) *src*, the identity of the original sender of the broadcast and (2) *age*, the number of times this message has been already broadcast.

Process  $p_i$  uses the following local variables: (1) *recv\_from<sub>i</sub>*, a set that serves as a buffer to contain the *quorum* currently being formed, and (2)  $\Sigma_i$ , the *quorum* returned by the algorithm.

---

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


---

```

1: init  $recv\_from_i \leftarrow \{p_i\}; \Sigma_i \leftarrow \perp$ 
2:
3: task T1: repeat forever
4:   broadcast( $p_i, 1$ )
5: end task
6:
7: task T2: upon reception of message( $src, age$ ) from  $p_j$ 
8:    $recv\_from_i \leftarrow recv\_from_i \cup \{src\}$ 
9:   if  $|recv\_from_i| \geq \alpha$  then
10:     $\Sigma_i \leftarrow recv\_from_i; recv\_from_i \leftarrow \{p_i\}$ 
11:   end if
12:   if  $age < \alpha - 1$  then broadcast( $src, age + 1$ ) end if
13: end task

```

---

**Algorithm Description.** Initially, the return value  $\Sigma_i$  of process  $p_i$  is  $\perp$ .

The algorithm keeps two parallel tasks:  $T1$  is an infinite loop that simply keeps broadcasting query messages at least once every  $\gamma$  units of time;  $T2$  handles any incoming message.

Every process identity received in a message by  $T2$  is simply added to the *quorum* buffer  $recv\_from_i$ . If the latter contains  $\alpha$  identities or more, then its size ensures a valid *quorum*. In this case,  $p_i$  saves the recent computed *quorum* and resets the quorum buffer in order to start the computing of the next *quorum* (lines 11 and 12).

The received message is then rebroadcast unless it has already been broadcast  $\alpha - 1$  times.

The mechanism of message expiration requires a *TVG* of class  $5-(\alpha, \gamma)'$ , which implies *direct journeys*. In the algorithm, every non faulty process rebroadcasts the messages it receives: if the next edge of the journey was not active yet, as it might happen in class  $5-(\alpha, \gamma)$ , the only way the message could “wait” would be to locally rebroadcast it in the meantime, which would imply in increasing its age. As a result, the message might not *reach* the number of processes necessary for each process to form a *quorum*.

### 6.3 A Round-Based Algorithm for $\Sigma_{\perp,k}$

Algorithm 2 presents an implementation of  $\Sigma_{\perp,k}$  requiring a *TVG* of class  $5-(\alpha, \gamma)$ . Since it does not make use of the message expiration mechanism, it does not require class  $5-(\alpha, \gamma)'$  and relies on asynchronous rounds to eliminate old messages from *faulty* processes. Its implementation uses a query-response mechanism.

**Notations.** A message contains the following parameters: (1)  $src$ : the identity of the original sender of the broadcast, (2)  $qr$ : the identities of every process which received the message and retransmitted it, (3)  $mid\_src$ : the timestamp of the *quorum* that the last query issued by  $src$  is attempting to form (corresponding to the  $mid_i$  of process  $src$ ).

Besides  $recv\_from_i$  and  $\Sigma_i$ , similarly to Algorithm 1,  $p_i$  keeps the following two other local variables:

- $mid_i$ : a round counter used to timestamp every *quorum* formed by process  $p_i$ . Every time a new *quorum* is completed,  $mid_i$  is incremented.
- $last\_known_i$ : the knowledge  $p_i$  has of the round counter of other processes. It is used to prevent  $p_i$  from uselessly rebroadcasting outdated messages. The variable contains tuples

---

**Algorithm 2** Implementation of  $\Sigma_{\perp, k}$  with asynchronous rounds for process  $p_i$ 


---

```

1: init  $mid_i \leftarrow 0$ ;  $last\_known_i \leftarrow \emptyset$ ;  $recv\_from_i \leftarrow \{p_i\}$ ;  $\Sigma_i \leftarrow \perp$ 
2:
3: task T1: repeat forever
4:   broadcast( $p_i$ ,  $\{p_i\}$ ,  $mid_i$ )
5: end task
6:
7: task T2: upon reception of message( $src$ ,  $qr$ ,  $mid\_src$ ) from  $p_j$ 
8:   if  $src = p_i$  and  $mid\_src = mid_i$  then ▷ RESPONSE
9:      $recv\_from_i \leftarrow recv\_from_i \cup qr$ 
10:    if  $|recv\_from_i| \geq \alpha$  then
11:       $\Sigma_i \leftarrow recv\_from_i$ ;  $recv\_from_i \leftarrow \{p_i\}$ 
12:       $mid_i \leftarrow mid_i + 1$ 
13:    end if
14:  else if  $src \neq p_i$  then ▷ QUERY
15:    if  $\langle src, last\_mid \rangle \in last\_known_i$ 
16:      and  $last\_mid \leq mid\_src$  then
17:      if  $last\_mid < mid\_src$  then
18:        replace in  $last\_known_i$   $\langle src, last\_mid \rangle$ 
19:        with  $\langle src, mid\_src \rangle$ 
20:      end if
21:      broadcast( $src$ ,  $qr \cup \{p_i\}$ ,  $mid\_src$ )
22:    else if  $\langle src, - \rangle \notin last\_known_i$  then
23:       $last\_known_i \leftarrow last\_known_i \cup$ 
24:       $\{\langle src, mid\_src \rangle\}$ 
25:      broadcast( $src$ ,  $qr \cup \{p_i\}$ ,  $mid\_src$ )
26:    end if
27:  end if
28: end task

```

---

of the form  $\langle p_j, mid_j \rangle$  where  $p_j$  is a process identity and  $mid_j$  is the last round number known by  $p_i$  for  $p_j$ .

**Algorithm Description.** Initially, the return value  $\Sigma_i$  of process  $p_i$  is  $\perp$ .

The algorithm keeps two parallel tasks:  $T1$  is an infinite loop that simply keeps broadcasting query messages at least once every  $\gamma$  units of time.  $T2$  handles any incoming message, considering two main cases:

Case 1: if the source process identity contained in the message is  $p_i$ , then the message is considered as a response to a query previously broadcast by  $p_i$ . If the round number is the current one, then the identities contained in the message are added to the *quorum* buffer  $recv\_from_i$ . If the latter contains  $\alpha$  identities or more, its size ensures a valid *quorum* (line 10). The task then saves the recently computed *quorum*, resets the quorum buffer  $recv\_from_i$ , and increments the round number in order to start computing a new *quorum* (lines 11 – 12).

Case 2: if the source process identity is different from  $p_i$ , then the message is considered as a query broadcast by another process which  $p_i$  needs to forward and respond to. If  $last\_known_i$  contains a more recent round number than the one received by  $p_i$  in the message,  $p_i$  ignores the message. Otherwise, it updates  $last\_known_i$  (lines 19 and 24) and broadcasts the same message with its own identity added in the  $qr$  parameter of the message (lines 21 and 25). This broadcast

acts both as a response mechanism and as a forward of the query which allows other processes to respond to it.

## 6.4 Comparison between Algorithms 1 and 2

Although the two algorithms solve the same problem, they present different strengths. Algorithm 2 does not make the strong assumption of *direct journeys*, as opposed to Algorithm 1. However, Algorithm 2 has the inconvenient of requiring two-way *recurrent connectivity*. Algorithm 1 also has the advantage of simplicity.

## 6.5 Proof of Correctness of Algorithms 1 and 2

**Theorem 2.** *Algorithms 1 and 2 ensure the intersection property of  $\Sigma_{\perp,k}$ .*

*Proof.* By lines 9 and 10 of Algorithm 1 and lines 10 and 11 of Algorithm 2, a process only returns *quorums* of size  $\geq \alpha$ .

If Assumption 1 holds, then it follows that any two *quorums* using the same *winning quorum*  $Q_{wi}$  intersect, since each *quorum* contains at least  $\lfloor \frac{|Q_{wi}|}{2} \rfloor + 1$  process identities taken from  $Q_{wi}$ . Given that there are only  $k$  different *winning quorums*, at least two out of any  $k + 1$  *quorums* contain the majority of the same *winning quorum* and therefore intersect.

If Assumption 2 holds, then it follows trivially that at least two out of  $k + 1$  *quorums* intersect since both contain  $\lfloor \frac{|Q_w|}{k+1} \rfloor + 1$  process identities taken from  $Q_w$ .  $\square$

**Lemma 1.** *Every correct process executing Algorithm 1 forms a new quorum infinitely often.*

*Proof.* Our algorithm imposes a size of at least  $\alpha$  for every *quorum*: as a result, a process may not be able to form new *quorums* after some time. Thus, every *correct* process must be reached by  $\alpha$  processes infinitely often. The message expiration mechanism can prevent a process from receiving messages from  $\alpha$  processes, even in a class  $5-(\alpha, \gamma)$  *TVG*, simply because a message can age needlessly by waiting for an edge to be activated. As a result, the  $(\alpha, \gamma)$ -*direct recurrent receiver connectivity* of class  $5-(\alpha, \gamma)$ ' is needed to ensure that Algorithm 1 will form *quorums* infinitely often. This class guarantees *direct  $\gamma$ -journeys*, which means that messages can cross all the necessary edges without waiting.  $\square$

**Lemma 2.** *Every correct process executing Algorithm 2 forms a new quorum infinitely often.*

*Proof.* Since it uses a query-response mechanism, Algorithm 2 requires every *correct* process to reach and be reached by  $\alpha$  processes infinitely often. This is exactly the guarantee provided by the  $(\alpha, \gamma)$ -*recurrent connectivity* property of *TVG* class  $5-(\alpha, \gamma)$ . 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 messages from  $\alpha$  infinitely often.  $\square$

**Lemma 3.** *There is a time  $\tau \in \mathcal{T}$  after which every new quorum formed by Algorithms 1 and 2 contains only correct processes.*

*Proof.* By definition, *faulty* processes will crash or leave the system in a finite time. Messages arrive in a finite time and in the case of Algorithm 1, are rebroadcast a finite number of times. Therefore with Algorithm 1, there is a time after which no information related to *faulty* processes remains in the system. Since processes form new *quorums* from fresh messages infinitely often

(Lemma 1), there is a time after which every new *quorum* formed will contain only *correct* processes.

For Algorithm 2, let  $t \in \mathcal{T}$  be the time at which the last *faulty* process crashes or leaves the system: 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.  $\square$

**Theorem 3.** *Algorithms 1 and 2 ensure the completeness property of  $\Sigma_{\perp,k}$ .*

*Proof.* The proof for *completeness* follows directly from Lemmas 1, 2 and 3.  $\square$

## 6.6 Necessity of $(\alpha, \gamma)$ -recurrent connectivity and $(\alpha, \gamma)$ -direct recurrent receiver connectivity

**Theorem 4.** *The  $(\alpha, \gamma)$ -recurrent connectivity property is a necessary requirement for Algorithm 2 and the  $(\alpha, \gamma)$ -direct recurrent receiver connectivity is a necessary requirement for Algorithm 1.*

*Proof.* Let us assume a TVG model that does not belong to class 5- $(\alpha, \gamma)$  or class 5- $(\alpha, \gamma)'$ . By definition, in such a model there is a *correct* process  $p_i$  that is unable to communicate with  $\alpha$  *correct* processes infinitely often. Thus, there is a time  $\tau \in \mathcal{T}$  at which  $p_i$  will stop forming new *quorums*, since both algorithms need  $\alpha$  processes to form a *quorum*.

Let  $p_j$  be a *faulty* process. We can imagine a run where  $p_j$  broadcasts a message before crashing, and  $p_i$  uses this message, and therefore includes  $p_j$ 's identity in its last formed *quorum* before  $\tau$ . As a result  $p_i$  will forever keep a *faulty* process in its *quorum*, thus violating the *completeness* property of  $\Sigma_{\perp,k}$ .  $\square$

## 7 An Algorithm for $k$ -Set Agreement in Dynamic Networks Using $\Sigma_{\perp,z}$

Algorithm 3 is derived from the algorithm presented by Bouzid and Travers in [7]. Similarly to the original one, processes are partitioned into  $z + 1$  disjoint sets of processes  $A_1, \dots, A_{z+1}$  such that  $\forall i, j, i \neq j, A_i \cap A_j = \emptyset; \bigcup A_i = \Pi$ ;

$\forall i \in [1..z] |A_i| = \lfloor \frac{n}{z+1} \rfloor; |A_{z+1}| = \lfloor \frac{n}{z+1} \rfloor + n \bmod(z+1)$ .

The original algorithm requires  $\Sigma_z$  with  $k \geq n - \lfloor \frac{n}{z+1} \rfloor$  while ours relies on  $\Sigma_{\perp,z}$  (with the same value for  $k$ ). Our algorithm requires the following assumption to hold:

**Assumption 3** (Quorum recurrent connectivity).  $\forall i, j \in \mathcal{C}$ , if the  $\Sigma_{\perp,z}$  *quorums* formed by  $i$  and  $j$  intersect infinitely often, then there is a recurrent connectivity from  $i$  to  $j$ :  $\forall \tau \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}_{(i,j)}^\tau : \text{departure}(\mathcal{J}) > \tau$ .

Similarly to Algorithms 1 and 2,  $\gamma$  is the maximum delay between two broadcasts of the same message by a process. We also require processes to receive their own broadcast messages within  $\gamma$  units of time.

Note that many implementations of  $\Sigma_{\perp,z}$  ensure *quorum recurrent connectivity*, as is the case with Algorithm 2. Therefore, Assumption 3 consists in relying on the assumptions already made to implement the failure detector instead of introducing new ones. This is not an additional

assumption compared to the original algorithm: the connectivity it implies is still weaker than the full recurrent connectivity of a static network.

For the conception of the new version of the algorithm, we have had to cope with the following constraints of the original algorithm: (1) The use of one-time broadcasts and eventual reception of the message by other processes. To fulfill this in a *TVG*, processes need to rebroadcast received messages. (2) The use of a selective multicast (line 1 of the original algorithm). We can translate this mechanism in a dynamic network by having every process broadcasting the message to their respective current neighbors while destination processes filter it and may not deliver it. (3) The assumption that every process knows, from the start, the identifiers of those processes which belong to its own partition. In the new version of the algorithm, every process knows, at the start, just the number of the partition to which it belongs and, during the run, it dynamically gets knowledge of its partition membership (or part of it).

Intuitively, at the algorithm initialization, every process has its own proposed value and whenever a process decides, it broadcasts the decided value. Notice that processes do not stop after deciding but must keep broadcasting their decision.

---

**Algorithm 3** Solving  $k$ -set agreement with  $\Sigma_{\perp,z}$ :  $p \in A_i$

---

```

1: init  $v \leftarrow$  initial value;  $dec \leftarrow false$ ;  $known \leftarrow \{p\}$ 
2:
3: task T0: repeat every  $\gamma$ 
4:   if  $dec$  then broadcast DEC( $v$ )
5:   else broadcast VAL( $p, i, v$ ) end if
6: end task
7:
8: task T1: upon reception of message VAL( $src, j, w$ )
9:   if  $dec \neq true$  then
10:    if  $i > j$  then
11:       $v \leftarrow w$ ;  $dec \leftarrow true$ ; decide( $v$ )
12:    else if  $i = j$  then
13:       $known \leftarrow known \cup \{src\}$ 
14:    end if
15:    broadcast VAL( $src, j, w$ )
16:  end if
17: end task
18:
19: task T2: repeat  $X \leftarrow \Sigma_{\perp,z}$  until  $X \subseteq known$ 
20:   if  $dec \neq true$  then
21:      $dec \leftarrow true$ ; decide( $v$ )
22:   end if
23: end task
24:
25: task T3: upon reception of message DEC( $w$ )
26:   if  $dec \neq true$  then
27:      $v \leftarrow w$ ;  $dec \leftarrow true$ ; decide( $v$ )
28:   end if
29: end task

```

---

**Notations.** The algorithm uses the following local variables: (1)  $v$ : variable that contains the value that process  $p$  expects to decide, or has decided, (2)  $dec$ : a boolean variable indicating whether  $p$  has decided or not, thus preventing  $p$  from deciding twice, (3)  $known$ : the current knowledge that  $p$  has of the processes in partition  $A_i$ . We denote  $\Sigma_{\perp,z}$  the process identities (or special value  $\perp$ ) returned by the  $\Sigma_{\perp,z}$  failure detector.

Two types of messages are defined: (1)  $DEC(v)$ : A message indicating that the sender has decided value  $v$ . Any process receiving such a message will immediately decide  $v$  in turn, unless it has already decided. (2)  $VAL(p, i, v)$ : A message indicating that a process  $p$  of partition  $A_i$  proposed the value  $v$ . Processes of partitions  $A_j$  with  $j < i$  will ignore this message, but a process of a higher partition receiving the message will immediately decide  $v$  unless it has already decided, and a process of partition  $A_i$  receiving this message will add  $p$  to its set  $known$ .

**Algorithm Description.** Initially,  $v$  is set to the initial value proposed by  $p$  and  $dec$  is set to false. The algorithm keeps four parallel tasks:

Task  $T0$  keeps broadcasting messages at least once every  $\gamma$  units of time. Before  $p$  has decided ( $dec = false$ ), it broadcasts  $VAL(p, i, v)$  messages informing processes in higher partitions of its initial value  $v$  and processes of partition  $A_i$  that they belong to the same partition as  $p$ . After  $p$  has decided ( $dec = true$ ),  $T0$  broadcasts  $DEC(v)$  messages informing neighbors of the decided value.

Tasks  $T1$  and  $T3$  handle the reception of messages of type  $VAL(\cdot)$  and  $DEC(\cdot)$  respectively.  $T3$  checks if  $p$  has already decided and if not,  $p$  decides the received value, updates  $v$  and switches  $dec$  to  $true$ .  $T1$  only handles  $VAL(\cdot)$  messages coming from lower (or the same) partitions. If the receiver process is in a higher partition and has not decided yet, then it acts exactly as just described for  $T3$ . If the sender of the message ( $src$ ) belongs to the same partition of  $p$  ( $A_i$ ),  $p$  adds  $src$  to  $known$ . Unless  $p$  has already decided, it always rebroadcasts the  $VAL(\cdot)$  messages it receives.

Finally,  $T2$  is a safety mechanism that allows decision in the case when all correct processes are in the same partition. This case is detected using  $\Sigma_{\perp,z}$  and  $known$ , which is  $p$ 's estimation of the membership of partition  $A_i$ . When and if it occurs,  $p$  just decides its initial value and switches  $dec$  to  $true$ .

In order to prevent a process from deciding multiple values, the deciding blocks (lines 10–11, 20–21 and 26–27) need to be executed atomically.

## 7.1 Proof of Correctness of Algorithm 3

The proof strongly relies on the proof of the original algorithm provided in [7]. Nevertheless, a few adaptations and additional assumptions needed to be made.

**Lemma 4** (Validity). *A decided value is a proposed value.*

*Proof.* The proof for validity is trivial and similar to the one presented for the original algorithm in [7]. A process decides either its own proposed value (line 21) or a value proposed by another process (lines 11 and 27).  $\square$

**Lemma 5** (Termination). *Every correct process eventually decides a value.*

*Proof.* For any *correct* process  $p$ , let  $R_p$  be the set of *correct* processes such that  $R_p = \{q \in \mathcal{C} \mid \forall \tau \in \mathcal{T}, \exists \tau_p, \tau_q > \tau : qr_p^{\tau_p} \cap qr_q^{\tau_q} \neq \emptyset\}$ . Note that  $p \in R_p$ , therefore  $R_p$  cannot be empty. It follows from Assumption 3 that there is *recurrent connectivity* within each set  $R_p$ .

If one *correct* process  $p$  decides, it will continuously broadcast a  $DEC(\cdot)$  message. All processes in  $R_p$  will eventually receive this message, thus deciding in turn. Hence, it is sufficient to prove that one process in each  $R_p$  group decides.

For every *correct* process  $p$ , let  $m_p = \max\{i \mid A_i \cap R_p \neq \emptyset\}$ . We consider two cases:

- $\forall i \neq m_p, A_i \cap R_p = \emptyset$ . Since there is *recurrent connectivity* within  $R_p$ , all the processes in  $A_{m_p} \cap R_p$  will eventually receive each other's  $VAL(\cdot)$  messages. Hence, there exists a time after which  $\forall q \in A_{m_p} \cap R_p, R_p = A_{m_p} \cap R_p \subseteq \textit{known}$ . It then follows from the *completeness* property of  $\Sigma_{\perp, z}$  and the definition of  $R_p$  that each  $q \in A_{m_p} \cap R_p$  will eventually exit the repeat loop of task  $T2$  and decide.
- $\exists l \neq m_p : A_l \cap R_p \neq \emptyset$ . Let  $r \in A_l \cap R_p$  and  $q \in A_{m_p} \cap R_p$ .  $r$  will send a  $VAL(\cdot)$  message which will be received by  $q$  at some point thanks to the *recurrent connectivity* within  $R_p$ . Since  $m_p > l$ ,  $q$  will decide upon reception of the message.

□

**Lemma 6** (Agreement). *At most  $k$  different values are decided.*

*Proof.* In order to account for the difference between  $\Sigma_z$  and  $\Sigma_{\perp, z}$ , we need to take the convention that  $\forall i, \perp \notin A_i$ . The proof for agreement in [7] relies only on the properties of the chosen partitioning of processes (which is the same as the one used for our algorithm),  $\Sigma_z$ , and the assumption that  $k \geq n - \lfloor \frac{n}{z+1} \rfloor$ . Most importantly, no assumption is made on graph connectivity, which allows the original proof to hold for our algorithm as well. The usage of *known* instead of  $A_i$  in task  $T2$  cannot possibly break the *agreement* property, because we ensure by construction that  $\textit{known} \subseteq A_i$ . □

**Theorem 5.** *Algorithm 3 solves  $k$ -set agreement*

*Proof.* The proof follows from Lemmas 4, 5 and 6 □

## 8 Conclusion

This paper provided an algorithm to solve the  *$k$ -set agreement* problem in dynamic networks with asynchronous communications, where both system membership and the communication graph evolve over time.

To this end we extended class 5 (*recurrent connectivity*) of  $TVG$ , defined a new failure detector  $\Sigma_{\perp, k}$ , and provided two implementations for the latter.

## References

- [1] Marcos Kawazoe Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, 2004.
- [2] Luciana Arantes, Fabíola Greve, Pierre Sens, and Véronique Simon. Eventual leader election in evolving mobile networks. In *OPODIS*, volume 8304, pages 23–37, 2013.
- [3] Abdulkader Benchi, Pascale Launay, and Frédéric Guidic. Solving consensus in opportunistic networks. In *ICDCN*, pages 1:1–1:10, 2015.
- [4] Martin Biely, Peter Robinson, and Ulrich Schmid. Agreement in Directed Dynamic Networks. In *SIROCCO*, volume 7355, pages 73–84, 2012.

- 
- [5] Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully Degrading Consensus and  $k$ -Set Agreement in Directed Dynamic Networks. *CoRR*, abs/1501.02716, 2015.
  - [6] François Bonnet and Michel 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*, volume 5873, pages 149–164, 2009.
  - [7] Zohir Bouzid and Corentin Travers. (anti- $\Omega^x \times \Sigma_z$ )-Based  $k$ -Set Agreement Algorithms. In *OPODIS*, volume 6490, pages 189–204, 2010.
  - [8] Jiannong Cao, Michel Raynal, Corentin Travers, and Weigang Wu. The eventual leadership in dynamic mobile networking environments. In *PRDC*, pages 123–130, 2007.
  - [9] Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. A Strict Hierarchy of Dynamic Graphs for Shortest, Fastest, and Foremost Broadcast. *CoRR*, abs/1210.3277, 2012.
  - [10] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *IJPEDES*, 27(5):387–408, 2012.
  - [11] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *JACM*, 43(4):685–722, 1996.
  - [12] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, 43(2):225–267, 1996.
  - [13] Soma Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Inf. Comput.*, 105(1):132–158, 1993.
  - [14] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *JACM*, 57(4), 2010.
  - [15] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC'04*, pages 338–346, 2004.
  - [16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, 1988.
  - [17] Afonso Ferreira. Building a reference combinatorial model for MANETs. *IEEE Network*, 18(5):24–29, 2004.
  - [18] Carlos Gómez-Calzado, Arnaud Casteigts, Alberto Lafuente, and Mikel Larrea. A Connectivity Model for Agreement in Dynamic Systems. In *Euro-Par*, 2015.
  - [19] Carlos Gómez-Calzado, Alberto Lafuente, Mikel Larrea, and Michel Raynal. Fault-Tolerant Leader Election in Mobile Dynamic Distributed Systems. In *PRDC*, pages 78–87, 2013.
  - [20] Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *STOC*, pages 513–522, 2010.
  - [21] Fabian Kuhn, Yoram Moses, and Rotem Oshman. Coordinated consensus in dynamic networks. In *PODC*, pages 1–10, 2011.

- [22] Fabian Kuhn and Rotem Oshman. Dynamic networks: models and algorithms. *SIGACT News*, 42(1):82–96, 2011.
- [23] Achour Mostéfaoui, Eric Mourgaya, and Michel Raynal. Asynchronous implementation of failure detectors. In *DSN*, pages 351–360, 2003.
- [24] Thibault Rieutord, Luciana Arantes, and Pierre Sens. Détecteur de défaillances minimal pour le consensus adapté aux réseaux inconnus. In *Algotel*, 2015.
- [25] Adam Sealfon and Aikaterini Sotiraki. Agreement in Partitioned Dynamic Networks. *CoRR arXiv:1408.0574*, 2014.



**RESEARCH CENTRE  
PARIS – ROCQUENCOURT**

Domaine de Voluceau, - Rocquencourt  
B.P. 105 - 78153 Le Chesnay Cedex

Publisher  
Inria  
Domaine de Voluceau - Rocquencourt  
BP 105 - 78153 Le Chesnay Cedex  
inria.fr

ISSN 0249-6399