A Failure Detector for k-Set Agreement in Dynamic Systems

Elise Jeanneau*, Thibault Rieutord*†, Luciana Arantes* and Pierre Sens*
*Sorbonne Universités, UPMC Univ Paris 06, CNRS, Inria, LIP6
Email: {denis.jeanneau, thibault.rieutord, luciana.arantes, pierre.sens}@lip6.fr
†ENS Rennes

***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.**

I. 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 static ones. In this paper, we are interested in a particular agreement problem, namely *k-set agreement* [1], 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 different values. It cannot be solved in asynchronous systems prone to f failures when $k \leq f$. To circumvent this impossibility, solutions enriched with failure detectors have been used. Failure detectors [2] are distributed oracles that provide processes with unreliable information on processes failures. They address questions such as the minimal information about failures needed to solve agreement problems.

In asynchronous message passing model with n nodes and $f < n$, it has been proved in [3] that the weakest failure detector for solving *consensus* (1-*set agreement*) is the class of eventual leader failure detector, denoted Ω, combined with the quorum failure detector, denoted Σ [4]. For *k-set agreement*, the weakest failure detector in message passing systems is unknown but the failure detector Σ_k was proved to be necessary [5].

Recently, Rieutord et al. [6] proposed the failure detector $\Sigma_\perp$, which is an adaptation of Σ [4] 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 Σ_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 [7]. In this article, the authors also 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 Σ_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 [8], solving *k-set agreement* in static networks with known membership.

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

II. Related Work

Aside from the failure detector approach of [2], other models have been proposed to solve agreement problems. Dwork et

This work was performed within the Labex SMART, supported by French state funds managed by the ANR within the Investissements d'Avenir programme under reference ANR-11-LABX-65.

al. introduced in [9] partially synchronous systems, where communication delays are bounded but the bound is not initially known by processes. Fetzer proposed in [10] the MCM model, where received messages are assumed to be correctly classified as "slow" or "fast". Robinson and Schmid proposed in [11] the message driven Asynchronous Bounded-Cycle model.

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 [12], Aguilera presents different system models where an infinite number of processes can join the system during a given run. Cao et al. presented in [13] 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 [12] and communicate through the support stations. Ferreira introduced in [14] a model, namely *Evolving Graphs*, in which the communication graph evolves over time. Based also on communication graph evolution, Casteigts et al. presented in [7] 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 III. Kuhn et al. proposed in [15] 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 [16] a dynamic model, which is close to the model of Kuhn et al. [15], but based on directed graphs and weaker connectivity assumptions. A survey of dynamic system models can be found in [17] and [18].

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. [15] is provided in [19], along with an algorithm for *simultaneous consensus* (all the nodes decide in the same synchronous round) and another one for Δ-*coordinated consensus* (all the nodes decide within Δ rounds of each other). An algorithm for *consensus* is also presented in [16], 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 [20]. 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 Ω [21], which is necessary to solve *consensus*, have been adapted for dynamic systems in [13], [18], [22]. The algorithm in [18] relies on partial synchrony assumptions, whereas the ones in [13], [22] are asynchronous and use message pattern assumptions.

Very few articles have attempted to solve the k-*set agreement* problem in dynamic networks. Exploiting the formalism of [15] and considering an upper bound on the number of processes, Sealfon and Sotiraki present in [23] an algorithm solving k-*set agreement* in *partitioned dynamic networks*. Biely et al. in [24] proposed in their model [16] 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.

III. Models and Definitions

A. Process Model

The system is made up of a finite set of n processes denoted $\Pi = \{p_1, p_2, ..., p_n\}$. 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 when 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 [9].

Processes follow the *n-arrival model* of [12]: 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.

B. Communication Model

Time-Varying Graph. We model the system dynamics using the formalism of the *Time-Varying Graph (TVG)* [7]. 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)* $\rho : E \times \mathcal{T} \to \{0, 1\}$ *is the edge presence function, indicating whether a given edge* $e \in E$ *is active at a given time* $t \in \mathcal{T}$*, (5)* $\zeta : E \times \mathcal{T} \to \mathbb{N}$ *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)* $\psi : V \times \mathcal{T} \to \{0, 1\}$ *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 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 ζ to express communication constraints, processes do not have access to

ζ 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 in the *TVG* formalism.

Definition 2 (Journey)**.** *A journey is a sequence of couples* $\mathcal{J} = \{(e_1, t_1), (e_2, t_2), ..., (e_m, t_m)\}$ *such that* $\{e_1, e_2, ..., e_m\}$ *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 $departure(\mathcal{J})$ and $t_m + \zeta(e_m, t_m)$ is denoted $arrival(\mathcal{J})$. 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 ($arrival(\mathcal{J}) - departure(\mathcal{J})$).

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 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 the edge is active for the entire time of the message transfer, a message sent infinitely often will be received infinitely often. 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 (we do not require channels to be FIFO).

TVG classes. Several classes of *TVGs* have been defined in [7] with respect to temporal properties on the network dynamics. 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:*
$\forall u, v \in V, \forall t \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}^*_{(u,v)} : departure(\mathcal{J}) > t$.

C. The k-Set Agreement Problem

The problem of *k-set agreement* was introduced by Chaudhuri in [1] 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.

IV. A New Failure Detector: $\Sigma_{\perp,k}$

The *quorum failure detector* Σ [4] 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* Σ_k [5] 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 Σ_k was proved in [5] to be necessary for solving *k-set agreement* in asynchronous systems, it seems natural to implement it using the *TVG* formalism as a stepping stone towards dynamic *k-set agreement*. However, the intersection property of Σ_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 $k = 1$ was proposed in [6] by the $\Sigma_\perp$ failure detector. The detector can return the special value $\perp$ 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 Σ_k to propose a failure detector which can be used to solve k-set agreement in dynamic asynchronous systems.

At any time, the $\Sigma_{\perp,k}$ detector provides each process with either the default value $\perp$ or a set of process identities (*quorum*). The output of the detector on process p at time t is denoted $\Sigma_{\perp,k,p}(t)$. The following properties must be satisfied:

Intersection. Out of any $k+1$ non-$\perp$ *quorums*, at least two intersect: $\forall t_1, ..., t_{k+1} \in \mathcal{T}, \forall p_1, ..., p_{k+1} \in \Pi, \exists i, j :$
$1 \leq i \neq j \leq k+1, (\Sigma_{\perp,k,p_i}(t_i) \neq \perp \wedge \Sigma_{\perp,k,p_j}(t_j) \neq \perp)$
$\implies \Sigma_{\perp,k,p_i}(t_i) \cap \Sigma_{\perp,k,p_j}(t_j) \neq \emptyset$.

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 Σ_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, the detector outputs $\perp$ forever on crashed processes.

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

V. New *TVG* Classes for Reliable Message Transmission

As described in Section III, *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 [25], 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 [26], Gómez-Calzado et al. consider that the assumption of instantaneous edge detection is too strong. Instead, they define β-*journeys*, in which edges are guaranteed to stay active at least β time, with β strictly longer than the message transfer time. The authors consider *synchronous* communications and therefore define β-*journeys* using an upper bound on the transfer time. To overcome these drawbacks, we define γ-*journeys*.

Similarly to β-*journeys*, γ-*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 (γ-Journey)**.** *A* γ-journey $\mathcal{J}$ *(where* $\gamma > 0$ *is a time duration) is a* journey *such that every node on the path can wait up to* γ *units of time after the next edge becomes active before forwarding the message. Since the message may be sent at any time during the* γ *time window and the channel latency may vary during that time, the edge must remain active long enough for the worst case duration:*
1) $\forall i, 1 \leq i \leq |\mathcal{J}|, e_i$ *stays active from time* t_i *until, at least, time* $t_i + max_{0 \leq j \leq \gamma}\{j + \zeta(e_i, t_i + j)\}$.
2) $\forall i, 1 \leq i < |\mathcal{J}|, t_{i+1} \geq t_i + max_{0 \leq j \leq \gamma}\{j + \zeta(e_i, t_i + j)\}$.

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

Definition 6 (Class 5-γ: γ-*recurrent connectivity*)**.** *TVGs of class 5-*γ *ensure that every process can reach every other process infinitely often through* γ-journeys*:*
$\forall u, v \in V, \forall t \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}^{\gamma}_{(u,v)} : departure(\mathcal{J}) > t$.

Class 5-γ is strictly stronger than class 5. Trivially, $\forall \gamma_1, \gamma_2 : \gamma_1 \geq \gamma_2 \Rightarrow$ class 5-$\gamma_1 \subseteq$ class 5-γ_2 (class 5 would be equivalent to "class 5-0", if γ could be equal to 0).

Class 5-γ' (γ-*direct recurrent connectivity*) is defined similarly, except that it uses $\mathcal{J}^{d\gamma}_{(u,v)}$ instead of $\mathcal{J}^{\gamma}_{(u,v)}$. It is significantly stronger than class 5-γ: the assumption of recurrent *direct journeys* reduces the dynamics of the system.

Although classes 5-γ and 5-γ' 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-(α, γ): (α, γ)-*recurrent connectivity*)**.** *Every correct process can reach and be reached through* γ-journeys *infinitely often by at least* α *correct processes.*
$\forall p_i \in \mathcal{C}, \exists P_i \subseteq \mathcal{C}, |P_i| \geq \alpha, \forall t \in \mathcal{T}, \forall p_j \in P_i, \exists \mathcal{J}_i \in \mathcal{J}^{\gamma}_{(p_i,p_j)} : departure(\mathcal{J}_i) \geq t \ \wedge \exists \mathcal{J}_j \in \mathcal{J}^{\gamma}_{(p_j,p_i)} : departure(\mathcal{J}_j) \geq t.$

Definition 8 (Class 5-$(\alpha, \gamma)'$: (α, γ)-*direct recurrent receiver connectivity*)**.** *Every correct process can be reached through* direct γ-journeys *infinitely often by at least* α *correct processes.*
$\forall p_i \in \mathcal{C}, \exists P_i \subseteq \mathcal{C}, |P_i| \geq \alpha, \forall t \in \mathcal{T}, \forall p_j \in P_i, \exists \mathcal{J} \in \mathcal{J}^{d\gamma}_{(p_j,p_i)} : departure(\mathcal{J}) \geq t.$

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

VI. Algorithms for $\Sigma_{\perp,k}$ in Dynamic Networks

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

The principle of both algorithms is the following: each process broadcasts their identity regularly and, infinitely often, waits for messages from α 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-(α, γ) 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 α that fits the definition of our *TVG* classes, but the processes are required to know the same α value ($\alpha \leq |\mathcal{C}|$). γ 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 γ units of time. Note that the bound value of γ 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.

A. 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. [27].

The message pattern model consists in assuming some properties on the relative order of message deliveries. When a certain number of messages (α) 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 avoids 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 [6] applied to k-*set agreement*.

Assumption 1 (Multiple winning quorums)**.**
$\exists Q_{w1}, Q_{w2}, ..., Q_{wk} \subseteq \Pi : \forall p \in \Pi$*, 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* α *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* α *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. Moreover, it can be shown that for every $\alpha \geq 2$ and $k \geq 2$, both assumptions are incomparable and none is weaker than the other one.

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 Σ_k in static networks in [8].

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

B. 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-$(\alpha, \gamma)'$.

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

```
1:  init recv_from_i ← {p_i}; Σ_i ← ⊥
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 ← recv_from_i ∪ {src}
9:      if |recv_from_i| ≥ α then
10:         Σ_i ← recv_from_i; recv_from_i ← {p_i}
11:     end if
12:     if age < α − 1 then broadcast(src, age + 1) end if
13: end task
```

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) $recv\_from_i$, a set that serves as a buffer to contain the *quorum* currently being formed, and (2) Σ_i, the *quorum* returned by the algorithm.

Algorithm Description. Initially, the return value Σ_i of process p_i is $\perp$. The algorithm keeps two parallel tasks: $T1$ is an infinite loop that keeps broadcasting query messages at least once every γ 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 α 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-(α, γ)', which implies *direct journeys*. Every process rebroadcasts the messages it receives: if the next edge of the journey was not active yet, as it might happen in class 5-(α, γ), 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*.

C. A Round-Based Algorithm for $\Sigma_{\perp,k}$

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

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

```
1:  init  mid_i ← 0; last_known_i ← ∅; recv_from_i ← {p_i};
    Σ_i ← ⊥
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 ← recv_from_i ∪ qr
10:         if |recv_from_i| ≥ α then
11:             Σ_i ← recv_from_i; recv_from_i ← {p_i}
12:             mid_i ← mid_i + 1
13:         end if
14:     else if src ≠ p_i then                         ▷ QUERY
15:         if ⟨src, last_mid⟩ ∈ last_known_i
16:                 and last_mid ≤ mid_src then
17:             replace in last_known_i ⟨src, last_mid⟩
18:                 with ⟨src, mid_src⟩
19:             broadcast(src, qr ∪ {p_i}, mid_src)
20:         else if ⟨src, −⟩ ∉ last_known_i then
21:             last_known_i ← last_known_i ∪ {⟨src, mid_src⟩}
22:             broadcast(src, qr ∪ {p_i}, mid_src)
23:         end if
24:     end if
25: end task
```

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 Σ_i, similarly to Algorithm 1, p_i keeps the following two other local variables: (1) 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. (2) $last\_known_i$: 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

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 Σ_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 γ 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 α 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 18 and 21) and broadcasts the same message with its own identity added in the *qr* parameter of the message (lines 19 and 22). This broadcast acts both as a response mechanism and as a forward of the query which allows other processes to respond to it.

D. 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.

E. Proof of Correctness of Algorithms 1 and 2

Theorem 1. *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, the size of returned *quorums* is $\geq \alpha$.

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. ■

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

Proof: Our algorithm imposes a size of at least α 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 α processes infinitely often. The message expiration mechanism can prevent a process from receiving messages from α processes, even in a class 5-(α, γ) *TVG*, simply because a message can age needlessly by waiting for an edge to be activated. As a result, the (α, γ)*-direct recurrent receiver connectivity* of class 5-(α, γ)' is needed to ensure that Algorithm 1 will form *quorums* infinitely often. This class guarantees *direct* γ*-journeys*, which means that messages can cross all the necessary edges without waiting. ■

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 α processes infinitely often. This is exactly the guarantee provided by the (α, γ)*-recurrent connectivity* property of *TVG* class 5-(α, γ). 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 it stops being isolated. As a result, every *correct* process will receive messages from α infinitely often. ■

Lemma 3. *There is a time τ after which every new* quorum *formed by Algorithms 1 and 2 contains only correct processes.*

Proof: By definition, *faulty* processes will leave the system in a finite time. Messages arrive in a finite time and, with 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 τ. Therefore, every *quorum* being currently built at τ 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 τ contains only *correct* processes. ■

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

Proof: The proof follows directly from Lemmas 1, 2, 3. ■

F. Necessity of TVG classes 5-(α, γ) and 5-$(\alpha, \gamma)'$

Theorem 3. *The class 5-(α, γ) (resp. class 5-$(\alpha, \gamma)'$) is a necessary requirement for Algorithm 2 (resp. Algorithm 1).*

Proof: Let us assume a *TVG* model that does not belong to class 5-(α, γ) or class 5-$(\alpha, \gamma)'$. By definition, in such a model there is a *correct* process p_i that is unable to communicate with α *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 α 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 τ. As a result p_i will forever keep a *faulty* process in its *quorum*, thus violating the *completeness* property of $\Sigma_{\perp,k}$. ■

VII. 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 [8]. Similarly to the original one, processes are partitioned into $z+1$ disjoint sets of processes $A_1, ..., A_{z+1}$ 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 Σ_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: $\forall \tau \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}^{\gamma}_{(i,j)} : departure(\mathcal{J}) > \tau$.*

Similarly to Algorithms 1 and 2, γ 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 γ 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 the processes in its own partition. In the new version of the algorithm, at the start every process only knows the number of the partition to which it belongs and, during the run, 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.

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 it 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 $known$ set.

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

```
1:  init  v ← initial value; dec ← false; known ← {p}
2:
3:  task T0: repeat every γ
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 ≠ true then
10:         if i > j then v ← w; dec ← true; decide(v)
11:         else if i = j then known ← known ∪ {src} end if
12:         broadcast VAL(src, j, w)
13:     end if
14: end task
15:
16: task T2: repeat X ← Σ⊥,z until X ⊆ known
17:     if dec ≠ true then dec ← true; decide(v) end if
18: end task
19:
20: task T3: upon reception of message DEC(w)
21:     if dec ≠ true then v ← w; dec ← true; decide(v) end if
22: end task
```

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 γ 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(.) and DEC(.) 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(.) 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(.) 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 $true$.

To prevent a process from deciding multiple values, the deciding lines (10, 17 and 21) need to be executed atomically.

A. Proof of Correctness of Algorithm 3

The proof strongly relies on the proof of the original algorithm provided in [8]. Nevertheless, a few adaptations and additional assumptions needed to be made.

Theorem 4. *Algorithm 3 ensures the validity property.*

Proof: The proof for validity is trivial and similar to the one presented for the original algorithm in [8]. A process decides either its own proposed value (line 17) or a value proposed by another process (lines 10 and 21). ■

Theorem 5. *Algorithm 3 ensures the termination property.*

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 $DEC(.)$ message. All processes in R_p will eventually receive this message and decide in turn. Hence, it is sufficient to prove that one process in each R_p set decides.

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

Case 1: $\forall \boldsymbol{i \neq m_p, A_i \cap R_p = \emptyset}$. Since there is *recurrent connectivity* within R_p, all the processes in $A_{m_p} \cap R_p$ will eventually receive each other's $VAL(.)$ messages. Hence, there exists a time after which $\forall q \in A_{m_p} \cap R_p, R_p = A_{m_p} \cap R_p \subseteq known$. 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.

Case 2: $\exists \boldsymbol{l \neq m_p : A_l \cap R_p \neq \emptyset}$. Let $r \in A_l \cap R_p$ and $q \in A_{m_p} \cap R_p$. r sends a $VAL(.)$ message which q receives at some point thanks to *recurrent connectivity* within R_p. Since $m_p > l$, q decides upon reception of the message. ■

Theorem 6. *Algorithm 3 ensures the agreement property.*

Proof: In order to account for the difference between Σ_z and $\Sigma_{\perp,z}$, we need to take the convention that $\forall i, \perp \notin A_i$. The proof for agreement in [8] relies only on the properties of the chosen partitioning of processes (which is the same as the one used for our algorithm), Σ_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 $known \subseteq A_i$. ■

VIII. 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] S. Chaudhuri, "More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems," *Inf. Comput.*, vol. 105, no. 1, pp. 132–158, 1993.

[2] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *JACM*, vol. 43, no. 2, pp. 225–267, 1996.

[3] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The weakest failure detectors to solve certain fundamental problems in distributed computing," in *PODC'04*, 2004, pp. 338–346.

[4] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui, "Tight failure detection bounds on atomic object implementations," *JACM*, vol. 57, no. 4, 2010.

[5] F. Bonnet and M. Raynal, "Looking for the Weakest Failure Detector for k-Set Agreement in Message-Passing Systems: Is Π_k the End of the Road?" in *SSS*, vol. 5873, 2009, pp. 149–164.

[6] T. Rieutord, L. Arantes, and P. Sens, "Détecteur de défaillances minimal pour le consensus adapté aux réseaux inconnus," in *Algotel*, 2015.

[7] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *IJPEDS*, vol. 27, no. 5, pp. 387–408, 2012.

[8] Z. Bouzid and C. Travers, "(anti-$\Omega^x \times \Sigma_z$)-Based k-Set Agreement Algorithms," in *OPODIS*, vol. 6490, 2010, pp. 189–204.

[9] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *JACM*, vol. 35, no. 2, pp. 288–323, 1988.

[10] C. Fetzer, "The Message Classification Model," in *PODC*, 1998, pp. 153–162.

[11] P. Robinson and U. Schmid, "The Asynchronous Bounded-Cycle Model," in *SSS*, 2008, pp. 246–262.

[12] M. K. Aguilera, "A pleasant stroll through the land of infinitely many creatures," *SIGACT News*, vol. 35, no. 2, pp. 36–59, 2004.

[13] J. Cao, M. Raynal, C. Travers, and W. Wu, "The eventual leadership in dynamic mobile networking environments," in *PRDC*, 2007, pp. 123–130.

[14] A. Ferreira, "Building a reference combinatorial model for MANETs," *IEEE Network*, vol. 18, no. 5, pp. 24–29, 2004.

[15] F. Kuhn, N. A. Lynch, and R. Oshman, "Distributed computation in dynamic networks," in *STOC*, 2010, pp. 513–522.

[16] M. Biely, P. Robinson, and U. Schmid, "Agreement in Directed Dynamic Networks," in *SIROCCO*, vol. 7355, 2012, pp. 73–84.

[17] F. Kuhn and R. Oshman, "Dynamic networks: models and algorithms," *SIGACT News*, vol. 42, no. 1, pp. 82–96, 2011.

[18] 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.

[19] F. Kuhn, Y. Moses, and R. Oshman, "Coordinated consensus in dynamic networks," in *PODC*, 2011, pp. 1–10.

[20] A. Benchi, P. Launay, and F. Guidec, "Solving consensus in opportunistic networks," in *ICDCN*, 2015, pp. 1:1–1:10.

[21] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *JACM*, vol. 43, no. 4, pp. 685–722, 1996.

[22] L. Arantes, F. Greve, P. Sens, and V. Simon, "Eventual leader election in evolving mobile networks," in *OPODIS*, vol. 8304, 2013, pp. 23–37.

[23] A. Sealfon and A. Sotiraki, "Agreement in Partitioned Dynamic Networks," *CoRR arXiv:1408.0574*, 2014.

[24] 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.

[25] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro, "A Strict Hierarchy of Dynamic Graphs for Shortest, Fastest, and Foremost Broadcast," *CoRR*, vol. abs/1210.3277, 2012.

[26] C. Gómez-Calzado, A. Casteigts, A. Lafuente, and M. Larrea, "A Connectivity Model for Agreement in Dynamic Systems," in *Euro-Par*, 2015.

[27] A. Mostéfaoui, E. Mourgaya, and M. Raynal, "Asynchronous implementation of failure detectors," in *DSN*, 2003, pp. 351–360.