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

Élise Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

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

Denis Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

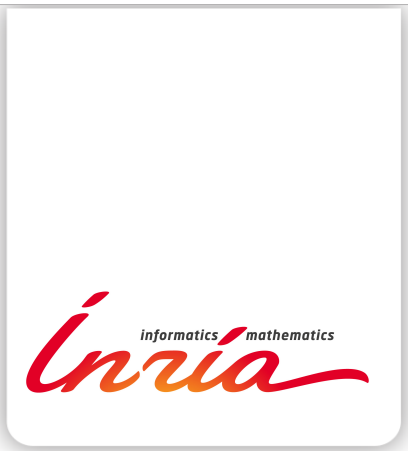
RESEARCH REPORT

N° 8727

May 2015

Project-Team Regal

ISSN 0249-6399 ISRN INRIA/RR--8727--FR+ENG



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

Denis Jeanneau, Thibault Rieutord, Luciana Arantes, Pierre Sens

Project-Team Regal

Research Report n° 8727 — May 2015 — 17 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. Exploiting the formalism of the Time-Varying Graph model, this paper proposes a new failure detector $\Sigma_{\perp,k}$, based on Σ_k and $\Sigma_\perp$, for solving k-set agreement in asynchronous dynamic networks. We present two algorithms that implement this new failure detector, making assumptions on the number of process failures and graph connectivity. We also provide an algorithm for solving k-set agreement using $\Sigma_{\perp,z}$, under an assumption on the relative values of k and z.

Key-words: k-set agreement, failure detectors, dynamic networks, time-varying graphs, asynchronous systems, 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 Asynchrones

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. En utilisant le formalisme du Time-Varying Graph, ce papier propose un nouveau détecteur de fautes $\Sigma_{\perp,k}$, basé sur Σ_k et $\Sigma_\perp$, résolvant le k-accord dans les réseau dynamiques asynchrones. Nous présentons deux algorithmes qui implémentent ce nouveau détecteur de fautes, sous réserve d'hypothèses sur le nombre de processus fautifs et la connectivité du graphe. Nous fournissons également un algorithme résolvant le k-accord avec $\Sigma_{\perp,k}$, en admettant une hypothèse sur les valeurs relatives de k et z.

Mots-clés : k-accord, détecteurs de défaillances, réseaux dynamiques, time-varying graphs, systèmes 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 vary over the time.

Agreement problems, the keystone of distributed computing problems, have been less studied in these new models than they have been in traditional ones. In this paper, we are interested in a particular agreement problem, namely *k-set agreement* [?], and how to solve it in asynchronous dynamic systems using failure detectors. 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* [?] are distributed oracle that provide processes with information, not always correct, on processes failures. They address questions on 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 [?] that the the weakest failure detector for solving consensus (i.e., 1-set agreement) is the class of eventual leader failure detector, denoted Ω, associated to the quorum failure detector, denoted Σ [?]. For k-set agreement ($k > 1$), the weakest failure detector in message passing systems is unknown but the failure detector Σ_k was proven to be necessary [?].

Recently, Rieutord et al. [?] proposed the failure detector $\Sigma_\perp$, which is an adaptation of Σ [?] 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.

Furthermore, 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 [?]. 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 asynchronous dynamic systems.

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 assumptions. The algorithms consider an upper bound $f < \frac{kn}{k+1}$ on the number of process failures, where n is the number of processes in the system.

4. An algorithm for solving *k-set agreement* in our model, using $\Sigma_{\perp,k}$ with $k = z\lfloor\frac{n}{z+1}\rfloor + (n \bmod (z+1))$.

The three algorithms presented in the current paper are extensions of the protocols proposed in [?], which were originally proposed to solve *k-set agreement* in static networks with known membership. Our solutions rely on assumptions on the number of process failures and extensions of *TVG* connectivity classes, making no assumptions on the timeliness of communications.

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 that we use. In Section 6 we present two algorithms implementing $\Sigma_{\perp,k}$ using different model assumptions. 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 [?], Aguilera presents different system models where an infinite number of processes can join the system during a given run.

Ferreira introduced in [?] a model, namely *Evolving Graphs*, in which the communication graph evolves over time. Based also on communication graph evolution, Casteigts et al. presented in [?] 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 [?] 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 [?] a dynamic model, which is close to Kuhn et al.'s model [?], but based on directed graphs and weaker connectivity assumptions.

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

Algorithms for Consensus and Associated Failure Detectors in Dynamic Systems. An algorithm for *consensus* in the model of Kuhn et al. [?] is provided in [?], along with an algorithm for *simultaneous consensus* (all the nodes decide in the same synchronous round) and a third one for Δ-*coordinated consensus* (all the nodes decide within Δ rounds of each other).

An algorithm for *consensus* is also presented in [?], consideing 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.

Implementations of the failure detector Ω [?] for dynamic systems are proposed in [?],[?]. The algorithm in [?] relies on partial synchrony assumptions, whereas the one in [?] is asynchronous and uses message pattern assumptions.

Algorithms for k-Set Agreement in Dynamic Systems. Exploiting the formalism of [?] and considering an upper bound on the number of processes, Sealfon and Sotiraki present in [?] an algorithm for solving *k-set agreement* in *partitioned dynamic networks*. Biely et al., based on the model introduced by themselves in [?], propose in [?] 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 asynchronous dynamic networks.

3 Models and Definitions

3.1 Process Model

The system is made up of a finite set of n processes denoted $\Pi = \{p_1, p_2, ..., p_n\}$. Processes in the system are uniquely identified, but are initially only aware of their own identity. There exists no bound on the relative speed of processes: the latter are, therefore, *asynchronous*.

A run is a sequence of steps executed by the processes while respecting the causality of operations (each message received has been previously sent).

Processes follow the *n-arrival model* of [?]: processes may join and leave the system, but only a total of n processes can join the system in a run. 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 $\mathcal{C}$, and the set of all *faulty* processes is denoted $\mathcal{F}$. We assume an upper bound f on the number of *faulty* processes in the system.

Although we make assumptions on the value of f, processes do not need to be aware of the value of n nor f. We just require them to know $n - f$.

3.2 Communication Model

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

Definition 1. *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)* $\mathcal{T} \subseteq \mathbb{N}$ *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 $\mathcal{T}$.

Note that, 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 using the formalism of the *TVG* model.

Definition 2. *A journey is a sequence of couples* $\mathcal{J} = \{(e_1, t_1), (e_2, t_2), ..., (e_k, t_m)\}$ *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. *A direct journey $\mathcal{J}$ is a journey such that every next edge in $\mathcal{J}$ is directly available. More formally:*

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

Channels. Processes communicate by sending each other messages using a *best-effort broadcast* primitive. The channels are *fair-lossy*, which means that messages may be lost, but, if a process p_i sends a message m to process p_j infinitely often such that, every time, the edge connecting p_i and p_j is active 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 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 [?] 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 (*recurrent connectivity*). More formally:

Definition 4. *TVGs of class 5 ensure that every process can reach every other process infinitely often:*

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

3.3 The k-Set Agreement Problem

The problem of *k-set agreement* was introduced by Chaudhuri in [?] 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* Σ [?] provides processes with *quorums* of process identities 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 [?] 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 proven in [?] 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 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 of Σ ($k = 1$) was proposed in [?] by the $\Sigma_\perp$ failure detector, which outputs either the normal output of Σ or the special value $\perp$. A process can return $\perp$ instead of a *quorum* to indicate that it has not yet gathered a minimal knowledge of the system to form a *quorum*. We apply the same concept to Σ_k in order to provide 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:

- **Intersection.**

$$\forall t_1, ..., t_{k+1} \in \mathcal{T}, \forall p_1, ..., p_{k+1} \in \Pi, \exists i, j : 1 \leq i \neq j \leq k+1,$$

$$(\Sigma_{\perp,k,p_i}(t_i) \neq \perp \wedge \Sigma_{\perp,k,p_j}(t_j) \neq \perp) \Rightarrow \Sigma_{\perp,k,p_i}(t_i) \cap \Sigma_{\perp,k,p_j}(t_j) \neq \emptyset \ .$$

- **Completeness.**

$$\exists \tau \in \mathcal{T}, \forall p \in \mathcal{C}, \forall t \in \mathcal{T}, t \geq \tau : \Sigma_{\perp,k,p}(t) \neq \perp \text{ and } \Sigma_{\perp,k,p}(t) \subseteq \mathcal{C} \ .$$

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 the minimal knowledge about process identities has not been reached, and therefore the intersection property is not expected to hold yet.

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 [?], is to assume that when an edge appears or disappears, the adjacent nodes are notified of this 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 [?], 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 transmission time. However, the authors consider *synchronous* communications and therefore define *β-journeys* using an upper bound on the transmission time. To overcome these drawbacks, we define *γ-journeys* as follows:

Definition 5. *A* γ-journey $\mathcal{J}$ *for* $\gamma > 0$ *is a* journey *such that:*

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

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

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

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

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 in Definition 5).

Class 5-γ' is significantly stronger than class 5-γ, as the assumption of recurrent *direct journeys* considerably 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.

Definition 7. ***Class 5***-(α, γ)*:* (α, γ)-recurrent connectivity

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

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

Definition 8. ***Class 5***-(α, γ)*':* (α, γ)-direct recurrent connectivity

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

$$\exists \mathcal{J}_1 \in \mathcal{J}^{d\gamma}_{(p_i,p_j)} : departure(\mathcal{J}_1) \geq t \wedge$$

$$\exists \mathcal{J}_2 \in \mathcal{J}^{d\gamma}_{(p_j,p_i)} : departure(\mathcal{J}_2) \geq t \ .$$

Class 5-(α, γ) ensures that any correct process can *reach* and be *reached* infinitely often by at least α correct processes, including itself. Note that although the formal definition only mentions one-way *journeys* from p_i to p_j, it is trivial to show that in a *non-directed* graph, if p_i can reach p_j infinitely often, then p_j can reach p_i infinitely often. Indeed, if p_i can reach p_j infinitely often, then there is a *non-directed* walk w in G between p_i and p_j such that every edge in w is active infinitely often. The same cannot be said for *direct journeys*, which is why class 5-(α, γ)' calls for a more exhaustive definition.

An alternative way to define class 5-(α, γ) and class 5-(α, γ)' would be to assume that there is a partition of $\mathcal{C}$ in subgraphs of size $\geq \alpha$ such that every subgraph belongs in class 5-γ and class 5-γ', respectively.

Note that for $\alpha = |\mathcal{C}|$, class 5-(α, γ) implies recurrent connectivity between all *correct* processes: we call this particular case *correct recurrent connectivity*.

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

In this section, we present two implementations of $\Sigma_{\perp,k}$ in the classes presented in 5, relying on assumptions on the number of *faulty* processes f (although processes themselves are only aware of $n - f$).

Both algorithms revolve around an idea introduced in [**?**]: if all the *quorums* returned by the algorithm contain at least $\lfloor \frac{n}{k+1} \rfloor + 1$ process ids, then out of any $k + 1$ such *quorums*, at least

two necessarily intersect, ensuring the intersection property of $\Sigma_{\perp,k}$. Thus, if $n-f \geq \lfloor \frac{n}{k+1} \rfloor + 1$, then the processes can simply broadcast a query message, and wait to receive $n-f$ responses to form a *quorum*. Hence, in order to prevent starvation, the system cannot suffer more than f failures, with $f < \frac{kn}{k+1}$.

Note that this failure pattern assumption could be replaced with a generalization of the message pattern assumption used to implement $\Sigma_\perp$ in [**?**], which would allow our algorithms to tolerate up to $n-1$ failures.

Bouzid and Travers assumed in [**?**] a static and complete communication graph with reliable links. To reproduce this same behavior, considering the *TVG* model, we need to cope with the following issues: (1) a process joining the system during the run may have missed the broadcast of other processes and (2) there is no guarantee that any process will have $n-f$ neighbors at any point in time, or even over time.

(1) is solved if processes constantly broadcast at regular intervals instead of just once. (2) is solved if every process rebroadcasts every message it receives, allowing nodes to communicate with all the nodes of the connected component which they are part of, instead of just their respective neighbors. However, such an approach is not enough, as that connected component may have fewer than $n-f$ nodes.

The worst case scenario is a run where every process stays isolated for the whole duration of the run. To circumvent such a scenario, we need to consider the following property in order to ensure the formation of *quorums* by correct processes: infinitely often, every *correct* process p_i needs to be able to reach and be reached by $n-f$ processes. It is not necessary that these processes are always the same but, given that the number of processes is finite, p_i will eventually be connected infinitely often with the same set of at least $n-f$ nodes. Furthermore, in order to satisfy the *completeness* property of $\Sigma_{\perp,k}$, this set must be composed by *correct* processes.

These assumptions are formalized in the properties of a *TVG* of class 5-(α, γ), where $\alpha = n-f$ and γ is the maximum delay between two broadcasts of the same message by a process. Note that, in this case, the bound value of γ implies that the relative speed of processes is finite. However, this value (and therefore the bound on the relative processes' speed) does not need to be known by processes. In fact, the knowledge of γ is only required when verifying whether a particular system fits our model assumptions. Therefore, we still consider processes to be *asynchronous* in our model.

We should also point out a third problem, induced by the use of infinite rebroadcast: 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. Aiming at overcoming this problem, we thus propose two algorithms based on different solutions. Both of them consider that $f < \frac{kn}{k+1}$, but the first algorithm requires a *TVG* of class 5-(α, γ)', whereas the second one requires a *TVG* of class 5-(α, γ).

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

Algorithm 1 provides *completeness* by limiting the number of times a message can be rebroadcast.

Notations. Broadcast messages contain the parameters: (1) *src*, the id 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:

- $recv\_from_i$: a set that keeps the identities of processes contained in the messages received by p_i since the last time it returned a *quorum*. When this variable has α values, it is considered the new *quorum*.

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

```
1:  Init
2:      recv_from_i ← ∅
3:      Σ_i ← ⊥
4:
5:  Task T1: repeat forever
6:      broadcast(p_i, 1)
7:
8:  Task T2: upon reception of message(src, age) from p_j
9:      recv_from_i ← recv_from_i ∪ {src}
10:     if |recv_from_i| ≥ α then
11:         Σ_i ← recv_from_i
12:         recv_from_i ← ∅
13:     end if
14:     if age < α − 1 then
15:         broadcast(src, age + 1)
16:     end if
17:
```

- 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 simply keeps broadcasting query messages; $T2$ 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)'. Class 5-(α, γ) is insufficient. In the algorithm, every non faulty process immediately 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*.

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

Algorithm 2 presents an implementation of $\Sigma_{\perp,k}$ tolerating fewer than $\frac{kn}{k+1}$ failures and 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 is quite similar to a query-response mechanism.

Notations. A message contains the following parameters: (1) *src*: the id of the original sender of the broadcast, (2) *qr*: the ids 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*).

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

```
1:  Init
2:      mid_i ← 0
3:      last_known_i ← ∅
4:      recv_from_i ← {p_i}
5:      Σ_i ← ⊥
6:
7:  Task T1: repeat forever
8:      broadcast(p_i, {p_i}, mid_i)
9:
10: Task T2: upon reception of message(src, qr, mid_src) from p_j
11:     if src = p_i and mid_src = mid_i then                          ▷ RESPONSE
12:         recv_from_i ← recv_from_i ∪ qr
13:         if |recv_from_i| ≥ α then
14:             Σ_i ← recv_from_i
15:             recv_from_i ← {p_i}
16:             mid_i ← mid_i + 1
17:         end if
18:     else if src ≠ p_i then                                         ▷ QUERY
19:         if ⟨src, last_mid⟩ ∈ last_known_i and last_mid ≤ mid_src then
20:             if last_mid < mid_src then
21:                 replace in last_known_i ⟨src, last_mid⟩ with ⟨src, mid_src⟩
22:             end if
23:             broadcast(src, qr ∪ {p_i}, mid_src)
24:         else if ⟨src, −⟩ ∉ last_known_i then
25:             last_known_i ← last_known_i ∪ {⟨src, mid_src⟩}
26:             broadcast(src, qr ∪ {p_i}, mid_src)
27:         end if
28:     end if
29:
```

Besides $recv\_from_i$ and Σ_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 of the form $\langle p_j, mid_j \rangle$ where p_j is a process id 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. $T2$ handles any incoming message, considering two main cases:

Case 1: if the source process id 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 ids 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 13). 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 14 – 16).

Case 2: if the source process id 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 21 and 25) and broadcasts the same message with its own identity added in the *qr* parameter of the message (lines 23 and 26). This broadcast acts both as a response mechanism and as a forward of the query which allows other processes to respond to it.

6.3 Comparison Between Algorithms 1 and 2

Although the two algorithms solve the same problems, they present different strengths. Algorithm 2 requires weaker connectivity assumptions and can therefore be applied in a larger number of systems than Algorithm 1, which needs *direct journeys*. However, Algorithm 2 has the inconvenient that not all useless messages are eliminated over the time: even though messages from old rounds are ignored by the original sender, they are still needlessly rebroadcast by every other process. Algorithm 1 also has the advantage of simplicity.

6.4 Proof of Correctness for Algorithms 1 and 2

6.4.1 *Intersection* Property

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

Proof. As made evident by lines 10 and 11 of Algorithm 1 and lines 13 and 14 of Algorithm 2, a process only returns *quorums* of size $\alpha = \lfloor \frac{n}{k+1} \rfloor + 1$. Thus, we can use the reasoning provided in [?]: any set of k+1 *quorums* contains a total of $(\lfloor \frac{n}{k+1} \rfloor + 1)(k+1) > n$ ids, and therefore contains at least one process id at least twice. The *intersection* property follows directly. □

6.4.2 *Completeness* Property of Algorithm 2.

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

Proof. This lemma is not trivial because our algorithm imposes a size of α for every *quorum* and a process may not be able to form new *quorums* after some time. Thus, every *correct* process must 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 2. *There is a time* $\tau \in \mathcal{T}$ *after which every new quorum formed contains only correct processes.*

Proof. By definition, *faulty* processes will crash or leave the system in a finite time. 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 remaining. Lemma 1 tells us that every remaining process 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. *Algorithm 2 ensures the completeness property of* $\Sigma_{\perp,k}$.

Proof. The proof for *completeness* follows directly from lemma 1 and lemma 2. □

6.4.3 *Completeness* Property of Algorithm 1.

The same reasonings of Algorithm 2 hold for algorithm 1, except for slight differences.

Lemma 3. *Every correct process forms a new quorum infinitely often.*

Proof. 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 connectivity* of class 5-(α,γ)' is needed to ensure that Algorithm 1 will form *quorums* infinitely often. □

Lemma 4. *There is a time* $\tau \in \mathcal{T}$ *after which every new quorum formed contains only* correct *processes.*

Proof. Faulty processes crash or leave the system in a finite time and messages arrive in a finite time and are rebroadcast a finite number of times. Therefore, 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, there is a time after which every new *quorum* formed will contain only *correct* processes. □

Theorem 3. *Algorithm 1 ensures the completeness property of* $\Sigma_{\perp,k}$.

Proof. The proof for completeness follows directly from lemma 3 and lemma 4. □

6.4.4 Necessity of (α,γ)-*recurrent connectivity.*

Theorem 4. *The* (α,γ)*-recurrent connectivity property is a necessary requirement for Algorithms 1 and 2 to ensure the properties of* $\Sigma_{\perp,k}$.

Proof. Let us assume a *TVG* model that does not belong in class 5-(α,γ). By definition, in such a model there is a *correct* process p_i that is unable to communicate with α *correct* processes infinitely often. From this we can deduce that 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 responds to a query from p_i before crashing, and p_i includes this response along with p_j's id 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 $\Sigma_{\perp,k}$. Therefore it is impossible for Algorithm 1 or Algorithm 2 to guarantee the *completeness* property in a model were (α,γ)-*recurrent connectivity* is not verified. □

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 [?]. 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 $k = z\lfloor \frac{n}{z+1} \rfloor + (n \bmod (z+1))$ while ours relies on $\Sigma_{\perp,z}$ (with the same value for k) as well as on *TVG* of class 5-(α, γ) where $\alpha = |\mathcal{C}|$ (*correct recurrent connectivity*) and γ is the maximum delay between two broadcasts of the same message by a process. Hence, 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 expectation of the eventual reception of the message by other processes. To fulfill this expectation 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 behavior in a dynamic network by having every process broadcasting the message to their respective current neighbors while the 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).

7.1 A Partition-based Algorithm for k-Set Agreement

Intuitively, at the algorithm initialization, every process has an initial 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 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 i proposed the value v. Processes of partitions $< i$ 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 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 naturally set to false. The algorithm keeps four parallel tasks:

Task $T0$ keeps broadcasting messages. Before p has decided ($dec = false$), it broadcasts VAL(p, i,v) messages informing processes in higher partitions ($> i$) of its initial value v and processes of partition i that they belong to the same partition of 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

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

```
1:  Init
2:      v ← initial value
3:      dec ← false
4:      known ← {p}
5:
6:  Task T0: repeat forever
7:      if dec then
8:          broadcast DEC(v)
9:      else
10:         broadcast VAL(p,i,v)
11:     end if
12:
13: Task T1: upon reception of message VAL(src, j, w)
14:     if dec ≠ true then
15:         if i > j then
16:             v ← w
17:             dec ← true
18:             decide(v)
19:         else if i = j then
20:             known ← known ∪ {src}
21:         end if
22:         broadcast VAL(src, j, w)
23:     end if
24:
25: Task T2: repeat X ← Σ⊥,z until X ⊆ known
26:     if dec ≠ true then
27:         dec ← true
28:         decide(v)
29:     end if
30:
31: Task T3: upon reception of message DEC(w)
32:     if dec ≠ true then
33:         v ← w
34:         dec ← true
35:         decide(v)
36:     end if
37:
```

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 every correct process is in the same partition. This case is detected using $\Sigma_{\perp,z}$ and the set *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 15-18, 26-28 and 32-35) need to be executed atomically.

7.2 Proof of Correctness for Algorithm 3

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

7.2.1 *Validity.*

Theorem 5. *Algorithm 3 ensures the validity property of k-set agreement.*

Proof. The proof for validity in [?] is trivial and similar to the one presented for the original algorithm. A process decides either its own proposal (line 28) or a value proposed by another process (lines 18, 35). □

7.2.2 *Termination.*

Theorem 6. *Algorithm 3 ensures the termination property of k-set agreement.*

Proof. For the *termination* property, the original proof makes connectivity assumptions which our algorithm does not make. The former assumes that if a *correct* process broadcasts a message, it will eventually be received by all *correct* processes. For this assertion to hold in a *TVG*, the *correct recurrent connectivity* property must be satisfied.

The main argument is that if a *correct* process never decides, then no *correct* process ever decide. Indeed, a deciding *correct* process will continuously broadcast a $DEC(.)$ message. Since links are fair-lossy and *correct recurrent connectivity* ensures that *correct* processes can always reach each other, all *correct* processes will eventually receive this message, and therefore decide. Therefore, it is sufficient to prove that one *correct* process will eventually decide.

Let $m = max\{i : A_i \cap \mathcal{C} \neq \emptyset\}$. We consider two cases:

- $\forall i \neq m, A_i \cap \mathcal{C} = \emptyset$. Thanks to *correct recurrent connectivity*, we know that every *correct* process in A_m will eventually receive the $VAL(.)$ messages of every other *correct* process in A_m. Hence, there exists a time after which $\forall p \in A_m, \mathcal{C} = A_m \cap \mathcal{C} \subseteq known$: the *completeness* property of $\Sigma_{\perp,z}$ then guarantees that each $p \in A_m \cap \mathcal{C}$ will eventually exit the repeat loop of task $T2$ and, therefore, decides.

- $\exists l \neq m : A_l \cap \mathcal{C} \neq \emptyset$. Let $p \in A_l \cap \mathcal{C}$ and $q \in A_m \cap \mathcal{C}$. p will send a $VAL(.)$ message which will be received by q at some point thanks to *correct recurrent connectivity*. Since $m > l$, q will decide upon reception of the message.

□

7.2.3 *Agreement.*

Theorem 7. *Algorithm 3 ensures the agreement property of k-set agreement.*

Proof. In order to account for the difference between Σ_z and $\Sigma_{\perp,z}$, we need to take the convention that $\forall i, \neg(\perp \in A_i)$. The proof for agreement in [?] only relies on the properties of the chosen partitioning of processes (which we used identically), Σ_z, and the assumption that $k = z\lfloor \frac{n}{z+1} \rfloor + (n \bmod (z+1))$. 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 $known \subseteq A_i$. □

8 Conclusion

This paper provided a *TVG*-based model and an algorithm to solve the *k-set agreement* problem in asynchronous dynamic networks 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 it.

Contrarily to our approach, several papers related to dynamic networks and evolving communication graph view the leaving of processes from the system as *changes* as opposed to *failures*. They, thus, assume that the number of changes should not be limited which is not compatible with the assumption made by our $\Sigma_{\perp,k}$ algorithms on the number of failures. Therefore, a future research could attempt to solve the *k-set agreement* problem in an asynchronous, wait-free dynamic environment, and rely on message pattern assumptions.

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 2013*, volume 8304, pages 23–37, 2013.

[3] Martin Biely, Peter Robinson, and Ulrich Schmid. Agreement in Directed Dynamic Networks. In *SIROCCO 2012*, volume 7355, pages 73–84, 2012.

[4] 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.

[5] François Bonnet and Michel 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 2009*, volume 5873, pages 149–164, 2009.

[6] Zohir Bouzid and Corentin Travers. (anti-$\Omega^x \times \Sigma_z$)-Based k-Set Agreement Algorithms. In *OPODIS 2010*, volume 6490, pages 189–204, 2010.

[7] 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.

[8] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *IJPEDS*, 27(5):387–408, 2012.

[9] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *J. ACM*, 43(4):685–722, 1996.

[10] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, 1996.

[11] Soma Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Inf. Comput.*, 105(1):132–158, 1993.

[12] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4), 2010.

[13] 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.

[14] Afonso Ferreira. Building a reference combinatorial model for MANETs. *IEEE Network*, 18(5):24–29, 2004.

[15] Carlos Gómez-Calzado, Arnaud Casteigts, Alberto Lafuente, and Mikel Larrea. A Connectivity Model for Agreement in Dynamic Systems. *Technical Report*.

[16] Carlos Gómez-Calzado, Alberto Lafuente, Mikel Larrea, and Michel Raynal. Fault-Tolerant Leader Election in Mobile Dynamic Distributed Systems. In *PRDC 2013*, pages 78–87, 2013.

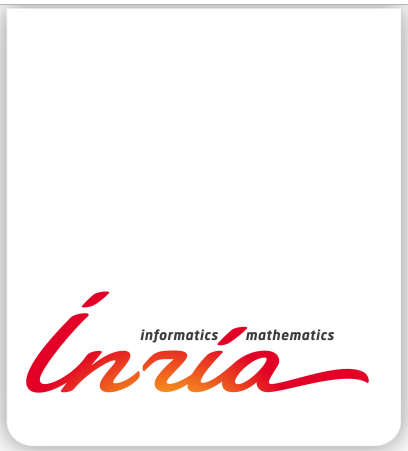
[17] Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *STOC 2010*, pages 513–522, 2010.

[18] Fabian Kuhn, Yoram Moses, and Rotem Oshman. Coordinated consensus in dynamic networks. In *PODC 2011*, pages 1–10, 2011.

[19] Fabian Kuhn and Rotem Oshman. Dynamic networks: models and algorithms. *SIGACT News*, 42(1):82–96, 2011.

[20] Thibault Rieutord, Luciana Arantes, and Pierre Sens. Détecteur de défaillances minimal pour le consensus adapté aux réseaux inconnus. In *Algotel*, 2015.

[21] 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