On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems *

Pierre Fraigniaud[1]**, Sergio Rajsbaum[2]***, and Corentin Travers[3]†

[1] CNRS and U. Paris Diderot, France.
Pierre.Fraigniaud@liafa.univ-paris-diderot.fr
[2] Instituto de Matemáticas, UNAM, D.F. 04510, Mexico.
rajsbaum@im.unam.mx
[3] CNRS and U. of Bordeaux, France.
travers@labri.fr

Abstract. Decentralized runtime monitoring involves a set of monitors observing the behavior of system executions with respect to some correctness property. It is generally assumed that, as soon as a violation of the property is revealed by any of the monitors at runtime, some recovery code can be executed for bringing the system back to a legal state. This implicitly assumes that each monitor produces a binary opinion, true or false, and that the recovery code is launched as soon as one of these opinions is equal to false. In this paper, we formally prove that, in a failure-prone asynchronous computing model, there are correctness properties for which there is no such decentralized monitoring. We show that there exist some properties which, in order to be monitored in a wait-free decentralized manner, inherently require that the monitors produce a number of opinions larger than two. More specifically, our main result is that, for every k, $1 \leq k \leq n$, there exists a property that requires at least k opinions to be monitored by n monitors. We also present a corresponding distributed monitor using at most $k+1$ opinions, showing that our lower bound is nearly tight.

1 Introduction

Runtime verification is concerned with monitoring software and hardware system executions. It is used after deployment of the system for ensuring reliability, safety, and security, and for providing fault containment and recovery. Its essential objective is to determine, at any point in time, whether the system is

* All authors are supported in part by the CONACYT-CNRS ECOS Nord M12M01 research grant.
** Additional support from the ANR project DISPLEXITY, and from the INRIA project GANG.
*** Additional support from UNAM-PAPIIT and LAISLA.
† Additional support from ANR project DISPLEXITY.

in a legal or illegal state, with respect to some specification. Consider a distributed system whose execution is observed by one or several monitors. Passing messages to a *central* monitor at every event leads to severe communication and computation overhead. Therefore, recent contributions [6,9,27] on runtime verification of distributed systems focused on *decentralized* monitoring, where a set of n monitors observe the behavior of the system. As soon as a violation of the legality of the execution is revealed by any of these monitors at runtime, recovery code can be executed for bringing the system back to a legal state. For example, the recovery code can reboot the system, or release its resources. This framework implicitly assumes that each monitor i produces a binary *opinion* $o_i \in \{\text{true}, \text{false}\}$, and that the recovery code is launched as soon as one of these opinions is equal to false. In this paper, we formally prove that, in a crash-failure prone asynchronous wait-free computing model [4], there are correctness properties for which such decentralized monitoring does not exist, even if we let the number of opinions grow to an arbitrary constant $k \geq 2$.

Let us consider the following motivating example arising often in practice [8], of a system in which *requests* are sent by clients, and *acknowledged* by servers. The system is in a legal state if and only if (1) all requests have been acknowledged, and (2) every received acknowledgement corresponds to a previously sent request. Each monitor i is aware of a subset R_i of requests that has been received by the servers, and a subset A_i of acknowledgements that has been sent by the servers. To verify legality of the system, each monitor i may communicate with other monitors in order to produce some opinion o_i. In the traditional setting of decentralized monitoring mentioned in the previous paragraph, it is required that the monitors produce opinions $o_i \in \{\text{true}, \text{false}\}$ such that, whenever the system is not in a legal state, at least one monitor produces the opinion false.

In runtime monitoring, a correctness property is described by a formula in some temporal logic. In this paper, we abstract away the logic, and directly specify the property by the set of *legal* configurations of the system, that we call a *distributed language*, denoted by $\mathcal{L}$. For instance, in the request-acknowledgement example above, $\mathcal{L}$ is the set of all configurations $\{(r_i, a_i),\ i \in I\}$ such that $\cup_{i \in I}\ r_i = \cup_{i \in I}\ a_i$, where $I \subseteq [1, n]$. Indeed, this language is specifying that all observed requests have been acknowledged, and every observed acknowledgement corresponds to a previously sent request. The monitors must produce opinions enabling to distinguish the legal configurations, i.e., those in $\mathcal{L}$, from the illegal ones. In order to make up their opinions, the monitors are able to communicate among themselves, so that each monitor can potentially collect system observations of other monitors. Since we are mostly interested in lower bounds, we ask very little from the monitors, and simply require that, for any pair (C, C') of configurations with $C \in \mathcal{L}$ and $C' \notin \mathcal{L}$, the *multiset* of opinions produced by the monitors given the legal configuration C must be different from the multiset of opinions given the illegal configuration C'.

In the centralized setting, more than two logical values may be required to avoid evaluating prematurely the correctness of a property that cannot be decided solely based on a prefix of the execution, like request-acknowledgement.

Hence [2,7] extended linear temporal logic (LTL) to logics with three values (e.g., {true, false, inconclusive}). More recently, it was recognised [8] that even three values are not sufficient to monitor some properties, and thus extensions of LTL with four logical values (e.g., {true, false, probably true, probably false}) were introduced. In this paper we argue that, in an asynchronous failure-prone decentralized setting, even four values may not be sufficient.

Our results. We consider decentralized monitoring in the *wait-free* setting [4]. (See Section 2 for details about this model, and for the reasons why we chose it). Our main result is a lower bound on the number of opinions to be produced by a runtime decentralized monitor in an asynchronous system where monitors may crash. This lower bound depends solely on the language, i.e., on the correctness property being monitored. More specifically, we prove that, for any positive integer n, and for any k, $1 \leq k \leq n$, there exists a distributed language requiring monitors to produce at least k distinct opinions in a system with n monitors. This result holds whatever the system does with the opinions produced by the monitors. That is, our lower bound on the number of opinions is inherent to the language itself — and not to the way the opinions are handled in order to launch the recovery code to be executed in case the system misbehaves.

The number of opinions required to runtime monitor languages in a decentralized manner is actually tightly connected to an intrinsic property of each language: its *alternation number*. This parameter essentially captures the number of times a sequence of configurations of the system alternates between legal and illegal. Our main result states that, for any k, $1 \leq k \leq n$, there exists a language with alternation number k which requires at least k opinions to be monitored by n monitors. This bound is essentially tight, as we also design a distributed monitor which, for any k, $1 \leq k \leq n$, and any distributed language $\mathcal{L}$ with alternation number k, monitors $\mathcal{L}$ using at most $k+1$ opinions in systems with n monitors.

Technically, in this paper, we establish a bridge between, on the one hand, runtime verification, and, on the other hand, distributed computability. Thanks to this bridge, we could prove our lower bound using arguments from (elementary) algebraic topology. More specifically, our impossibility result for 2 opinions is obtained using graph-connectivity techniques sharing similarities with the FLP impossibility result for consensus [15], while our general impossibility result uses higher-dimensional techniques similar to those used in set agreement impossibility results e.g. [22,23].

As far as we know, this paper is the first one studying necessary conditions for monitoring distributed systems with failures.

Related work. The main focus in the literature is on sequential runtime verification. The monitors are event-triggered [24], where every change in the state of the system triggers the monitor for analysis. There is work also in time-triggered monitoring [10], where the monitor samples the state of the program at regular time intervals. Parallel monitoring has been addressed in [20] to some extent by focusing on low-level memory architecture to facilitate communication between

application and analysis threads. The concept of separating the monitor from the monitored program is considered in, e.g., [28]. Later, [9] uses a specialized parallel architecture (GPU), to implement runtime formal verification in a parallel fashion. Efficient automatic signaling monitoring in multi-core processors is considered in [13].

Closer to our setting is decentralized monitoring. In sequential runtime verification one has to monitor the requirement based on a single behavioral trace, assumed to be collected by some global observer. A central observer basically resembles classical LTL monitoring. In contrast, in decentralized monitoring, there are several partial behavioural traces, each one collected at a component of the system. Intuitively, each trace corresponds to the view that the component has of the execution. In decentralized LTL monitoring [6] a formula ϕ is decomposed into local formulas, so monitor i evaluates locally ϕ_i, and emits a boolean-valued opinion. In our terminology, an "and interpretation" is used. That is, it is assumed a global violation can always be detected locally by a process. In addition, it is assumed the set of local monitors communicate over a synchronous bus with a global clock. The goal is to keep the communication among monitors minimal. In [26] the focus is in monitoring safety properties of a distributed program's execution, also using an "and interpretation". The decentralized monitoring algorithm is based on formulae written in a variant of past time LTL. For the specific case of relaxed memory models, [11] presents a technique for monitoring that a program has no executions violating sequential consistency. There is also work [19] that targets physically distributed systems, but does not focus on distributed monitoring.

To the best of our knowledge, the effects of asynchrony and failures in a decentralized monitoring setting were considered for the first time in [17]. We extend this previous work in two ways. First, we remove the restriction that the monitors can produce only two opinions. Second, [17] investigated applications to locality, while here we extend the framework and adapt it to be able to apply it to a more general decentralized monitoring setting.

Related work in the distributed computing literature includes seminal papers such as [12] for stable property detection in a failure-free message-passing environment, and [5] for distributed program checking in the context of self-stabilization.

Organization of this paper. The distributed system model is in Section 2. Distributed languages and wait-free monitoring are presented in Section 3. In Section 4 we present the example of monitoring leader election. Our main result is in Section 5. Its proof is presented in Section 6. We conclude the paper and mention some open problems in Section 7. A full version [18] provides additional details and all the proofs.

2 Distributed system model

There are many possible computation and communication models for distributed computation. Here we assume wait-free asynchronous processes that may fail by

crashing, communicating by reading and writing a shared memory. This model serves as a good basis to study distributed computability: results in this model can often be extended to other popular models, such as when up to a fixed number of processes can crash (in a dependent or independent way). Also, message-passing, or various networking models that limit direct process-to-process connectivity, are essentially computationally equivalent or less powerful than shared memory. We recall here the main features of the wait-free model, and refer to textbooks such as [4] for a more detailed description, as well as for the relation to other distributed computing models.

The asynchronous read/write shared memory model assumes a system consisting of n asynchronous processes. Let $[n] = \{1, \ldots, n\}$. We associate each process to an integer in $[n]$. Each process runs at its own speed, that may vary along with time, and the processes may fail by *crashing* (i.e., halt and never recover). We consider *wait-free* distributed algorithms, in which a process never "waits" for another process to produce some intermediate result. This is because any number of processes may crash (and thus the expected result may never be produced).

The processes communicate through a shared memory composed of atomic registers, organised as an array of n single-writer/multiple-reader (SWMR) registers, one per process. Register $i \in [n]$ supports the operation $read()$ that returns the value stored in the register, and can be executed by any process. It also support de operation $write(v)$ that writes the value v in the register, and can be executed only by process i.

In our algorithms we use a *snapshot* operation by which a process can read all n SWMR registers, in such a way that a snapshot returns a copy of all the values that were simultaneously present in the shared memory at some point during the execution of the snapshot operation (snapshots are linearizable). Snapshots can be implemented by a wait-free algorithm (any number of processes may crash) using only the array of n SWMR registers [1] (see also textbooks such as [25]). Thus, we may assume snapshots are available to the processes, without loss of generality. The algorithms are simplified, as well as the proofs of our theorems, without modifying the outcomes of our results.

In a *distributed algorithm* each process starts with an *input value*, repeats a loop N times, consisting of writing to its register, taking a snapshot and making local computations[4]. At the end each process produces an *output value*. In a *step*, a process performs an operation on the registers (i.e., writes or snapshots). A *configuration* completely describes the state of the system. That is, a configuration specifies the state of each register as well as the local state of each process. An *execution* is a (finite) sequence of alternating configurations and steps, starting and ending in a configuration. A process *participates* in an execution if it takes at least one step in the execution. We assume that the first step of a process is a write, and it writes its input.

[4] If the set of possible input values is finite, all processes may execute the loop the same number of times, N (e.g. see [3]).

3 Distributed languages and wait-free monitoring

3.1 Distributed languages

Let A be an alphabet of symbols, representing the set of possible values produced by some distributed algorithm to be monitored. Each process $i \in [n]$ has a read-only variable, $input_i$, initially equal to a symbol $\perp$ (not in A), and where the value to be monitored is deposited. We consider only the simplest scenario, where these variables change only once, from the value $\perp$, to a value in A. The goal is for the processes to monitor that, collectively, the values deposited in these variables are correct.

Formally, consider an execution $C_0, s_1, C_1, \ldots$, where each C_i is a configuration and each s_i is a step (write or snapshot) by some process, and C_0 is the initial configuration where all SWMR registers are empty. We assume the first step by a process i is to write its input, and is taken only once its variable $input_i$ is initialized to a value in A. Thus, s_1 is a write step by some process.

The correctness specification to be monitored is usually stated as a global predicate in some logic (e.g. [13,14]). We rephrase the predicate in terms of what we call a *distributed language*. An *instance* over alphabet A (we may omit A when clear from the context) is a set of pairs $s = \{(\mathrm{id}_1, a_1), \ldots, (\mathrm{id}_k, a_k)\}$, where $\{\mathrm{id}_1, \ldots, \mathrm{id}_k\} \subseteq [n]$ are distinct process identities, and $a_1, \ldots, a_k$ are (not necessarily distinct) elements of A. A distributed language $\mathcal{L}$ over the alphabet A is a collection of instances over A. Given a language $\mathcal{L}$, we say that an instance s is *legal* if $s \in \mathcal{L}$ and *illegal* otherwise.

Let $s = \{(\mathrm{id}_1, a_1), \ldots, (\mathrm{id}_k, a_k)\}$ be an instance over A. We denote by $\mathrm{ID}(s)$ the set of identities in s, $\mathrm{ID}(s) = \{\mathrm{id}_1, \ldots, \mathrm{id}_k\}$. The multiset of values in s is denoted by $val(s)$ (formally, a function that assigns to each $a \in A$ a non-negative integer specifying the number of times a is equal to one of the a_i in s).

Note that an instance s can describe an assignment of values from A to the input variables of a subset of processes. More precisely, consider an execution $C_0, s_1, C_1, \ldots, s_k, C_k,\ k \geq 1$. Suppose the processes that have taken steps in this execution are those in P, $P \subseteq [n]$. This execution defines the instance $s = \{(\mathrm{id}_1, a_1), \ldots, (\mathrm{id}_k, a_k)\}$ over A, where $\mathrm{ID}(s) = P$ and a_i is the first value written by process id_i. A configuration C_k also defines an instance, given by the input variables of processes that have written at least once (from the local state of a process, one can deduce if it has already executed a write operation).

An execution is *correct* if and only if its instance s is in $\mathcal{L}$. If the execution is correct, then processes in $\mathrm{ID}(s)$ have values as specified by the language (and the other processes have not yet been assigned a value or may be slow in announcing their values).

Consider for example the language `req-ack`, which captures a simplified version of the request-acknowledgment problem mentioned in the introduction, in which no more than q requests are sent by the clients. Requests and acknowledgments are identified with integers in $[q]$. A process id_i may know of some subset of requests $r_i \subseteq [q]$, and some subset of acknowledgments $a_i \subseteq [q]$. The language

`req-ack` over alphabet $A = 2^{[q]} \times 2^{[q]}$ is defined by instances s as follows

$$s = \Big\{(\mathrm{id}_1, (r_1, a_1)), \ldots, (\mathrm{id}_k, (r_k, a_k))\Big\} \in \texttt{req-ack} \iff \bigcup_{1 \le i \le k} r_i = \bigcup_{1 \le i \le k} a_i.$$

For each process i, the sets r_i and a_i denote the (possibly empty) sets of requests and acknowledgments, respectively, that process i is aware of. An instance is legal if and only if every request has been acknowledged.

As another example, consider *leader election*, for which it is required that one unique process be identified as the leader by all the other processes. This requirement is captured by the language `leader` defined over $A = [n]$ as follows:

$$s = \big\{(\mathrm{id}_1, \ell_1), \ldots, (\mathrm{id}_k, \ell_k)\big\} \in \texttt{leader} \iff \exists i \in [k] : \mathrm{id}_i = \ell_1 = \cdots = \ell_k. \quad (1)$$

An instance is legal if and only if all the processes agree on the identity ℓ of one of them.

3.2 Decentralized monitoring

Monitoring the correctness specified by a language $\mathcal{L}$ involves two components: an *opinion-maker* M, and an *interpretation* μ. The opinion-maker is a distributed algorithm executed by the processes enabling each of them to produce an individual *opinion* about the validity of the outputs of the system. We call the processes running this algorithm *monitors*, and the (finite) set of possible individual opinions U, the *opinion set*.

The interpretation μ specifies the way one should interpret the collection of individual opinions produced by the monitors about the validity of the monitored system. We use the minimal requirement that the opinions of the monitors should be able to distinguish legal instances from illegal ones according to $\mathcal{L}$. Consider the set of all multi-sets over U, each one with at most n elements. Then $\mu = (\mathbf{Y}, \mathbf{N})$ is a partition of this set. $\mathbf{Y}$ is called the "yes" set, and $\mathbf{N}$ is called the "no" set.

For instance, when $U = \{0, 1\}$, process may produce as an opinion either 0 or 1. Together, the processes produce a multi-set of at most n boolean values. We do not consider which process produce which opinion, but we do consider how many processes produce a given opinion. The partition produced by the AND-operator [17,16] is as follows. For every multi-set of opinions S, we set $S \in \mathbf{Y}$ if every opinion in S is 1, otherwise, $S \in \mathbf{N}$.

Given a language $\mathcal{L}$ over an alphabet A, a *monitor for* $\mathcal{L}$ is a pair (μ, M), as follows.

- The opinion-maker M is a distributed wait-free algorithm that outputs an opinion u_i at every process i. The input of process i is any element a_i of A (assigned to its read-only variable $input_i$). Each process i is required to produce an opinion u_i such that: (1) every non-faulty process eventually produces an output (termination), and (2) if process i outputs u_i, then we must have: $u_i \in U$ (validity).

– Consider any execution of M where all participating processes have decided an opinion. If the instance s corresponding to the execution is legal, i.e., $s \in \mathcal{L}$, the monitors must produce a multiset of opinions $S \in \mathbf{Y}$, and if the instance s is illegal, i.e., $s \notin \mathcal{L}$, then they must produce a multiset of opinions in $\mathbf{N}$.

The paper focusses on the following question: given a distributed language $\mathcal{L}$, how many opinions are needed to monitor $\mathcal{L}$?

3.3 Opinion and alternation numbers

As stated above, we are interested in the smallest size $|U|$ of the opinion set enabling the monitors, after the execution of some distributed algorithm, to output opinions that distinguish legal instances from illegal ones. Hence, we focus on the following parameter associated with every distributed language.

Definition 1 (Opinion number). *Let $\mathcal{L}$ be a distributed language on n processes. The* opinion number *of $\mathcal{L}$ is the smallest integer k for which there exists a monitor (μ, M) for $\mathcal{L}$ using a set of at most k opinions. It is denoted by* $\#\text{opinion}(\mathcal{L})$.

As we shall see, there are monitors using a small number of opinions, independent of the size of the alphabet used to define $\mathcal{L}$, and depending only on the number n of processes. The opinion number is shown to be related to a combinatorial property of languages, captured by the notion of *alternation number*. Given a language $\mathcal{L}$ over the alphabet A, the alternation number of $\mathcal{L}$ is the length of a longest increasing sequence of instances $s_1, \ldots, s_k$ with alternating legality. More formally:

Definition 2 (Alternation number). *Let $\mathcal{L}$ be a distributed language. The* alternation number *of $\mathcal{L}$ is the largest integer k for which there exists instances $s_1, \ldots, s_k$ such that, for every i, $1 \leq i < k$, $s_i \subset s_{i+1}$, and either $(s_i \in \mathcal{L}) \wedge (s_{i+1} \notin \mathcal{L})$ or $(s_i \notin \mathcal{L}) \wedge (s_{i+1} \in \mathcal{L})$. It is denoted by* $\#\text{altern}(\mathcal{L})$.

Clearly, the alternation number is at most n since an instance has at most n elements.

4 Monitoring leader election

As a warm up example, let us show that the language `leader` of Equation 1 can be monitored using three opinions, namely, that $\#\text{opinion}(\texttt{leader}) \leq 3$. To establish this result, we describe a monitor for `leader`, called *traffic-light*. The set of opinions consists of three values, namely $\{\text{red}, \text{orange}, \text{green}\}$. Recall that the input of each process $i \in [n]$ is a value ℓ_i where $\ell_i \in [n]$ is supposed to be the identity of the leader. The opinion maker works as follows. Each monitor i writes its identity and it own input ℓ_i in shared memory, and then reads the whole memory with a snapshot operation. The snapshot returns a set of pairs,

$s_i = \{(\mathrm{id}_j, \ell_j), j \in I\}$ for some I, that includes the values written so far in the memory. Recall that processes run asynchronously, hence a process may collect values from only a subset of all processes. Process i decides "green" if every process in s_i agrees on the same leader, and the ID of the common leader is the ID of one of the processes in s_i. Instead, if two or more processes in s_i have distinct leaders, then process i decides "red". In the somewhat "middle" case in which every process in s_i agrees on the same leader (i.e., same ID), but the ID of the common leader is not an ID of a process in s_i, then process i decides "orange".

More formally, the traffic-light opinion maker uses two procedures: "agree" and "valid". Given a set $s = \{(\mathrm{id}_1, \ell_1), \ldots, (\mathrm{id}_k, \ell_k)\}$ of pairs $(\mathrm{id}_i, \ell_i) \in [n] \times [n]$, agree($s$) is true if and only if $\ell_i = \ell_j$ for every i, j, $1 \leq i, j \leq k$. For a same s, valid(s) is true if and only if, for every ℓ_i, $1 \leq i \leq k$, there exists $j, 1 \leq j \leq k$ such that $\mathrm{id}_j = \ell_i$. Each process performs the pseudo-code below:

Opinion-maker at process p with input ℓ:
write $(\mathrm{ID}(p), \ell)$ to p's register ;
snapshot memory, to get $s = \{(\mathrm{id}_1, \ell_1), \ldots, (\mathrm{id}_k, \ell_k)\}$;
if agree(s) **and** valid(s) **then** decide "green"
else if agree(s) **but** not valid(s) **then** decide "orange" **else** decide "red".

The interpretation of the opinions produced by the monitors is the following. An opinion u_i produced by process i is an element of the set $U = \{\text{green, orange, red}\}$. The opinion-maker produces a multi-set u of opinions. We define the yes-set $\mathbf{Y}$ as the set of all multi-sets u with no red elements, and at least one green element. Hence, $\mathbf{N}$ is composed of all multi-sets u with at least one red element, or with no green elements.

Now, one can easily check that the traffic-light monitor satisfies the desired property. That is, for every set $s = \{(\mathrm{id}_1, \ell_1), \ldots, (\mathrm{id}_k, \ell_k)\}$ of pairs $(\mathrm{id}_i, \ell_i) \in [n] \times [n]$, if u denotes the multi-set of opinions produced by the monitors, then we have

$$s \in \texttt{leader} \iff u \in \mathbf{Y}.$$

Interestingly enough, one can prove that the language `leader` *cannot be monitored using fewer than three opinions.* Namely,

Proposition 1. #opinion(`leader`) = 3.

Crucially, the fact that three opinions are required, and that, in particular, the opinions true and false are not sufficient, is an inherent property of the language `leader`, independently of the opinion-maker algorithm, and independently of the interpretation of the opinions produced by the monitors. The lower bound argument enabling to establish this result is not hard but uses a fundamental theorem about two-process read/write wait-free computation: the graph of executions is connected (e.g. see [3]).

As we mentioned before, the number of opinions required to monitor a distributed language is strongly related to its alternation number. The sequence of

instances

$$s_1 = \{(1,2)\},\ s_2 = \{(1,2),(2,2)\},\ \text{and}\ s_3 = \{(1,2),(2,2),(3,3)\}$$

satisfies $s_1 \subset s_2 \subset s_3$. Moreover s_1 and s_3 are illegal, while s_2 is legal (as far as `leader` is concerned). We thus infer that the alternation number of `leader` is at least 3. In fact, it can be shown that its alternation number is exactly 3. Namely,

Proposition 2. $\#\text{altern}(\texttt{leader}) = 3$.

Intuitively, the alternation between legal and illegal instances forces the processes to use three opinions. Given s_1, process 1 may say that the instance is "probably illegal" (orange), while, given s_2, process 2 may say that the instance is "potentially legal" (green). Only process 3, given s_3, can declare that the instance is "definitively illegal" (red), no matter the number of further processes that may show up.

5 Main result

In this section, we state our main result, that is, a lower bound on the number of opinions needed to monitor languages with n monitors.

Theorem 1. *For any $n \geq 1$, and every k, $1 \leq k < n$, there exists a language $\mathcal{L}$ on n processes, with alternation number k, that requires at least k opinions to be monitored. For $k = n$, there exists a language $\mathcal{L}$ on n processes, with alternation number n, that requires at least $n+1$ opinions to be monitored.*

In other words, there are system properties which require a large number of opinions to be monitored. Before dwelling into the details of the proof of Theorem 1, we want to stress the fact that our lower bound is essentially the best that can be achieved in term of alternation number. Indeed, Theorem 1 says that, for every k, there exists a language $\mathcal{L}$ with alternation number k such that $\#\text{opinion}(\mathcal{L}) \geq \#\text{altern}(\mathcal{L})$. We show that this lower bound is essentially tight. Indeed, we establish the existence of a *universal* monitor that can monitor all distributed languages using a number of opinions equal roughly to the alternation number. More specifically, we show that, for every k, and for every language $\mathcal{L}$ with opinion number k, we have $\#\text{opinion}(\mathcal{L}) \leq \#\text{altern}(\mathcal{L}) + 1$.

Theorem 2. *There exists a monitor which, for every $k \geq 1$, monitors every language with alternation number k using at most $k+1$ opinions.*

Since the alternation number of a language on n processes is at most n, Theorem 2 yields the following.

Corollary 1. *There exists a monitor which, for every $n \geq 1$, monitors every language on n processes, using at most $n+1$ opinions. Moreover, this monitor uses at most $k+1$ opinions for every execution in which at most k processes participate.*

It is worth noticing that the monitor of Corollary 1 has an interpretation μ which does not depend at all on the language to be monitored, not even on the number of processes involved in the language. (The same holds for Theorem 2). The opinion-maker (as well as the one for Theorem 2), does however depend on the language, but only up to a limited extent. Indeed, the general structure of the opinion-maker is independent of the language. It simply uses a black box that returns whether $s \in \mathcal{L}$ for any instance s. Apart from this, the opinion-maker is essentially independent of the language. In this sense it is *universal*.

The full proof of Theorem 2 is omitted for lack of space. The rest of the paper is dedicated to describing the main ideas of the proof of our main result.

6 Orientation-detection tasks, and proof of Theorem 1

To establish our lower bound, we show that the design of distributed runtime monitors using few opinions is essentially equivalent to solving a specific type of tasks, that we call *orientation-detection* tasks. This equivalence is made explicit thanks to an *equivalence lemma* (Lemma 1). Introducing orientation-detection tasks requires elementary notions of combinatorial topology.

6.1 Tasks and combinatorial topology terminology

When solving a distributed task[5], each process starts with a private input value and has to eventually decide irrevocably on an output value. (In our setting, the input value of a process is a symbol in a given alphabet A, and the output value is an opinion). A process $i \in [n]$ is initially not aware of the inputs of other processes. Consider an execution where only a subset of k processes participate, $1 \leq k \leq n$. These processes have distinct identities $\{\mathrm{id}_1, \ldots, \mathrm{id}_k\}$, where for every $i \in [k]$, $\mathrm{id}_i \in [n]$. A set $s = \{(\mathrm{id}_1, x_1), \ldots, (\mathrm{id}_k, x_k)\}$ is used to denote the input values, or output values, in the execution, where x_i denotes the value of the process with identity id_i — either an input value (e.g., a symbol in a given alphabet A), or a output value (e.g., an opinion).

The monitor task. An opinion-maker M for a language $\mathcal{L}$ on n processes with opinion set U, and interpretation $\mu = (\mathbf{Y}, \mathbf{N})$ is a distributed wait-free algorithm that solves the following *monitor task*. Any instance over alphabet A is a possible assignment of inputs in A to the processes. If process i has input $a_i \in A$, then i is required to produce as output an opinion $u_i \in U$ such that, in addition to satisfying termination and validity, it also satisfy *consistency*, defined as follows. Consider any execution, where I is the set of processes that do not crash, and all others crash without taking any steps. Let $s = \{(\mathrm{id}_i, a_i), i \in I\}$, and let $u = \{u_i, i \in I\}$ denote the multiset of opinions that are eventually output by the processes in I. We must have: $s \in \mathcal{L} \iff u \in \mathbf{Y}$.

[5] A task is the basic distributed computing problem, defined by a set of inputs to the processes and for each input to the processes, a set of legal outputs of the processes – see, e.g., [22].

Simplices and complexes. Let s' be a subset of a "full" set $s = \{(1, x_1), \ldots, (n, x_n)\}$, i.e., a set s such that $\mathrm{ID}(s) = [n]$. Since any number of processes can crash, all such subsets s' are of interest for taking into account executions where only processes in $\mathrm{ID}(s')$ participate. Therefore, the set of possible input sets forms a *complex* because its sets are closed under containment. Similarly, the set of possible output sets also form a complex. Following the standard terminology of combinatorial topology, the sets of a complex are called *simplexes*. Hence every set s' as above is a simplex.

More formally, a *complex* $\mathcal{K}$ is a set of vertices $V(\mathcal{K})$, and a family of finite, nonempty subsets of $V(\mathcal{K})$, called *simplexes*, satisfying: (1) if $v \in V(\mathcal{K})$ then $\{v\}$ is a simplex, and (2) if s is a simplex, so is every nonempty subset of s. The *dimension* of a simplex s is $|s| - 1$, the dimension of $\mathcal{K}$ is the largest dimension of its simplexes, and $\mathcal{K}$ is *pure* of dimension k if every simplex belongs to a k-dimensional simplex. In distributed computing, the simplexes (and complexes) are often *chromatic*, since each vertex v of a simplex is labeled with a distinct process identity $i \in [n]$.

A *distributed task* T is formally described by a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ and $\mathcal{O}$ are pure $(n-1)$-dimensional complexes, and Δ is a map from $\mathcal{I}$ to the set of non-empty sub-complexes of $\mathcal{O}$, satisfying $\mathrm{ID}(t) \subseteq \mathrm{ID}(s)$ for every $t \in \Delta(s)$. We call $\mathcal{I}$ the input complex, and $\mathcal{O}$ the output complex. Intuitively, Δ specifies, for every simplex $s \in \mathcal{I}$, the valid outputs $\Delta(s)$ for the processes in $\mathrm{ID}(s)$ that may participate in the computation. We assume that Δ is (sequentially) computable.

Given any finite set U and any integer $n \geq 1$, we denote by $\mathit{complex}(U, n)$ the $(n-1)$-dimensional *pseudosphere* [22] complex induced by U: for each $i \in [n]$ and each $x \in U$, there is a vertex labeled (i, x) in the vertex set of $\mathit{complex}(U, n)$. Moreover, $u = \{(\mathrm{id}_1, u_1), \ldots, (\mathrm{id}_k, u_k)\}$ is a simplex of $\mathit{complex}(U, n)$ if and only if u is properly colored with identities, that is $\mathrm{id}_i \neq \mathrm{id}_j$ for every $1 \leq i < j \leq k$. In particular, $\mathit{complex}(\{0, 1\}, n)$ is (topologically equivalent) to the $(n-1)$-dimensional sphere. For $u \in \mathit{complex}(U, n)$, we denote by $\mathit{val}(u)$ the multiset formed of all the values in U corresponding to the processes in u.

6.2 Orientation-detection tasks

An *oriented complex*[6] is a complex whose every simplex s is assigned a sign, $\mathit{sign}(s) \in \{-1, +1\}$. Given an oriented input complex, $\mathcal{J}$, a natural task consists in computing distributively the sign of the actual input simplex. That is, each process is assigned as input a vertex of $V(\mathcal{J})$, and the set of all the vertices assigned to the processes forms a simplex $s \in \mathcal{J}$. Ideally, one would like that processes individually decide "yes" if the simplex is oriented $+1$ and "no" otherwise. However, this is impossible in general because processes do not have the same view of the execution, and any form of non-trivial agreement cannot be solved in a wait-free manner [15]. Thus, we allow processes to express their knowledge through values in some larger set U.

[6] In the case of chromatic manifolds, our definition is equivalent to usual definition of orientation in topology textbooks.

Definition 3 (Orientation detection task). *Let $\mathcal{J}$ be a $(n-1)$-dimensional oriented complex. A task $T = (\mathcal{J}, \mathcal{U}, \Delta)$, with $\mathcal{U} = \mathit{complex}(U, n)$ for some set U, is an* orientation detection *task for $\mathcal{J}$ if and only if for every two $s, s' \in \mathcal{J}$, and every $t \in \Delta(s)$ and $t' \in \Delta(s')$: $\mathit{sign}(s) \neq \mathit{sign}(s') \Rightarrow \mathit{val}(t) \neq \mathit{val}(t')$.*

Hence, to detect the orientation of a simplex s, the processes i, $i \in I \subseteq [n]$, occurring in a simplex s have to collectively decide a multiset $\mathit{val}(t) = \{\mathit{val}(i), i \in I\}$ of values in U, where $\mathit{val}(i)$ denotes the value decided by process i. If $\mathcal{J}$ is non-trivially oriented, i.e., if there exist $s, s' \in \mathcal{J}$ of the same dimension, with $\mathit{sign}(s) \neq \mathit{sign}(s')$, then no orientation-detection tasks for $\mathcal{J}$ exists with $|U| = 1$, because one must be able to discriminate the different orientations of s and s'. Instead, for every oriented complex $\mathcal{J}$, there exists an orientation-detection task for $\mathcal{J}$ with $|U| = 2$. To see why, consider the task $T = (\mathcal{J}, \mathcal{U}, \Delta)$, where $\mathcal{U}$ is the $(n-1)$-dimensional sphere, and Δ maps every k-dimensional simplex $s \in \mathcal{J}$ with $\mathit{sign}(s) = -1$ (resp., $+1$) to the k-dimensional simplex $t \in \mathcal{U}$ with $\mathit{val}(t) = \{0, 0, \ldots, 0\}$ (resp., $\mathit{val}(t) = \{1, 1, \ldots, 1\}$). However, this latter task is not necessarily wait-free solvable (i.e., solvable in our context of asynchronous distributed computing where any number of processes can crash). The complexity of detecting the orientation of an oriented complex $\mathcal{J}$ is measured by the smallest k for which there exists an orientation-detection task $T = (\mathcal{J}, \mathcal{U}, \Delta)$ that is wait-free solvable, with $\mathcal{U} = \mathit{complex}(U, n)$, and $|U| = k$.

In the next subsection, we show that the problem of finding the minimum-size set U for detecting the orientation of an arbitrary given oriented complex $\mathcal{J}$ is essentially equivalent to finding the minimum-size set of opinions U for monitoring a language $\mathcal{L}_{\mathcal{J}}$ induced by $\mathcal{J}$ (and its orientation).

6.3 Equivalence lemma

This section shows that the notion of monitoring and the notion of orientation-detection are essentially two sides of the same coin.

Let $\mathcal{L}$ be a n-process distributed language defined over an alphabet A. We define $\mathcal{J}_{\mathcal{L}} = \mathit{complex}(n, A)$. That is, for every collection $\{a_1, \ldots, a_k\}$ of at most k elements of A, $1 \leq k \leq n$, and every k-subset $\{\mathrm{id}_1, \ldots, \mathrm{id}_k\} \subseteq [n]$ of distinct identities, $\{(\mathrm{id}_1, a_1), \ldots, (\mathrm{id}_k, a_k)\}$ is a simplex in $\mathcal{J}_{\mathcal{L}}$. Let us orient $\mathcal{J}_{\mathcal{L}}$ as follows. For every simplex $s \in \mathcal{J}_{\mathcal{L}}$, we define:

$$\mathit{sign}(s) = \begin{cases} +1 \text{ if } s \in \mathcal{L}; \\ -1 \text{ otherwise.} \end{cases}$$

Conversely, let $\mathcal{J}$ be a well-formed oriented complex. We say that an oriented complex $\mathcal{J}$ on n processes is *well-formed* if for every $I \subseteq [n]$, there exists $s, s' \in \mathcal{J}$ with $\mathrm{ID}(s) = \mathrm{ID}(s') = I$ and $\mathit{sign}(s) = -\mathit{sign}(s')$. We set $\mathcal{L}_{\mathcal{J}}$ as the n-process language defined over the alphabet $A = \{+1, -1\} \times V(\mathcal{J})$. That is, each element of A is a pair (σ, v) where σ is a sign in $\{+1, -1\}$ and v a vertex of $\mathcal{J}$. The language $\mathcal{L}_{\mathcal{J}}$ is the set of instances $s = \{(\mathrm{id}_1, (\sigma_1, v_1)), \ldots, (\mathrm{id}_k, (\sigma_k, v_k))\}$ specified as follows:

$$s \in \mathcal{L}_{\mathcal{J}} \iff \begin{cases} t = \{(\mathrm{id}_1, v_1), \ldots, (\mathrm{id}_k, v_k)\} \text{ is a simplex of } \mathcal{J}, \\ \text{and } \mathit{sign}(t) = \sigma_i \text{ for every } i,\ 1 \leq i \leq k. \end{cases}$$

That is, in a legal instance, each process is assigned a vertex of some simplex $t \in \mathcal{J}$ together with the orientation of t.

We have now all ingredients to state formally the first main ingredient toward establishing Theorem 1: the equivalence between language-monitoring and orientation-detection.

Lemma 1 (Equivalence lemma).
• *Let $\mathcal{L}$ be a n-process language. If there exists $k \geq 1$ and a wait-free solvable orientation-detection task for $\mathcal{J}_{\mathcal{L}}$ using values in some set of size k, then there exists a monitor for $\mathcal{L}$ using at most k opinions.*
• *Let $\mathcal{J}$ be a well-formed oriented complex, and let $k \geq 1$. If no orientation-detection task for $\mathcal{J}$ is wait-free solvable using k values, then the language $\mathcal{L}_{\mathcal{J}}$ requires at least $k+1$ opinions to be monitored.*

The proof of Lemma 1 is omitted from this extended abstract. This lemma establishes an equivalence between wait-free solving orientation-detection tasks and monitoring a language with few opinions. It can be shown that, in addition, this lemma preserves alternation numbers in the following sense. The concept of alternation number (for languages) can be similarly defined for oriented complexes: for an oriented complex $\mathcal{J}$, the alternation number of $\mathcal{J}$ is the length of a longest increasing sequence of simplexes of $\mathcal{J}$ with alternating orientations. Formally:

Definition 4 (Alternation number of oriented complexes). *Let $\mathcal{J}$ be an oriented complex. The* alternation number*, $\#\text{altern}(\mathcal{J})$, of $\mathcal{J}$ is the largest integer k for which there exists $s_1, \ldots, s_k \in \mathcal{J}$ such that, for every i, $1 \leq i < k$, $s_i \subset s_{i+1}$ and $sign(s_i) \neq sign(s_{i+1})$.*

The equivalence established in Lemma 1 preserves alternation number as stated by the following result.

Lemma 2. *For every language $\mathcal{L}$, and every well-formed oriented complex $\mathcal{J}$, we have $\#\text{altern}(\mathcal{J}_{\mathcal{L}}) = \#\text{altern}(\mathcal{L})$ and $\#\text{altern}(\mathcal{L}_{\mathcal{J}}) \leq \#\text{altern}(\mathcal{J}) + 1$.*

6.4 Sketch of the proof of Theorem 1

Due to lack of space, we only sketch the proof of Theorem 1. We use the correspondence between monitors and orientation-detection tasks as stated in Lemma 1, and focus on orientation-detection tasks. Given $k, 1 \leq k < n$, we carefully build an oriented complex $\mathcal{J}$ with alternation number $k-1$ and shows that any orientation-detection task with input complex $\mathcal{J}$ cannot be solved wait-free with $k-1$ values or less. Therefore, by the equivalence Lemma (Lemma 1), the language $\mathcal{L}_{\mathcal{J}}$ induced by $\mathcal{J}$ requires at least k values to be monitored. To complete the proof, we establish that the alternation number of $\mathcal{L}_J$ satisfies $\#\text{altern}(\mathcal{L}_J) = \#\text{altern}(\mathcal{J}) + 1 = k$. (The case $k = n$ is similar, except that we construct $\mathcal{J}$ with alternation number n, and $\#\text{altern}(\mathcal{L}_{\mathcal{J}}) = \#\text{altern}(\mathcal{J})$.).

The main challenge lies in constructing, and orienting the complex $\mathcal{J}$ in such a way that no orientation-detection task with input $\mathcal{J}$ is wait-free solvable

with less than k values. One important ingredient in the proof is an adaptation of Sperner's Lemma to our setting. To get an idea of how the proof proceeds, consider a ℓ-dimensional simplex $s \in \mathcal{J}$ whose all $(\ell-1)$-dimensional simplexes have sign -1, but one which has sign $+1$. Assume moreover that ℓ values only are used to encode the signs of these faces. Recall that any wait-free distributed algorithm induces a mapping from a subdivision of the input complex to the output complex [23]. By Sperner's Lemma, we prove that, whatever the opinion-maker does, at least one ℓ-dimensional simplex s' resulting from the subdivision of s satisfies $|val(s')| = \ell + 1$. That is, $\ell + 1$ values are used to determine the orientation of s, for every monitor (μ, M). In the full paper we describe the many details omitted here, that are behind this intuition. □

7 Conclusions and future work

We investigated the minimum number of opinions needed for runtime monitoring in an asynchronous distributed system where any number of processes may crash. We considered the simplest case, where each process outputs a single value just once, and the monitors verify that the values collectively satisfy a given correctness condition. A correctness condition is specified by a collection of legal sets of these values, that may occur in an execution. Each monitor expresses its opinion about the correctness of the set of outputs, based on its local perspective of the execution. We proved lower bounds on the number of opinions, and presented distributed monitors with nearly the same number of opinions.

Many avenues remain open for future research. It would be interesting to derive a temporal logic framework that corresponds to ours, and that associates to opinions a formal meaning in the logic. In our setting the processes produce just one output and the monitors must verify that, collectively, the set of outputs produced is correct. It would of course be interesting to extend our results to the case where each process produces a sequence of output values. Also, opinions are anonymous. The interpretation specifies which multisets of opinions indicate a violation, independently of the identities of the monitors that output them. We do not know whether or not taking into account the identities would help reducing the total number of opinions needed. Finally, it would be interesting to extend our results to other models, such as t-resilient models in which not more than t processes may fail.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.
2. O. Arafat, A. Bauer, M. Leucker, and C. Schallhart. Runtime verification revisited. Technical Report TUM-I0518, Technischen Universität München, 2005.
3. H. Attiya and S. Rajsbaum. The Combinatorial Structure of Wait-Free Solvable Tasks. *SIAM J. Comput.*, 31(4):1286–1313, 2002.
4. H. Attiya and J. L. Welch. *Distributed computing: fundamentals, simulations and advanced topics.* Wiley, USA, 2004.

5. B. Awerbuch and G. Varghese. Distributed Program Checking: A Paradigm for Building Self-stabilizing Distributed Protocols (Extended Abstract). *SFCS*, pp. 258–267. IEEE, 1991.
6. A. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods*, lncs #7436, pp. 85–100. Springer, 2012.
7. A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. *FSTTCS*, lncs #4337, pp. 260–272. Springer, 2006.
8. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. and Comput.*, 20(3):651–674, 2010.
9. S. Berkovich, B. Bonakdarpour, and S. Fischmeister. Gpu-based runtime verification. *IPDPS*, pp. 1025–1036. IEEE, 2013.
10. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. *Formal Methods*, pp. 88–102. Springer, 2011.
11. J. Burnim, K. Sen, C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. *TACAS*, lncs#6605, pp. 11–25, 2011.
12. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75. ACM, 1985.
13. H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal. A distributed abstraction algorithm for online predicate detection. *SRDS*, pp. 101–110. IEEE, 2013.
14. R. Cooper and K. Marzullo. Consistent detection of global predicates. *Workshop on Parallel and Distributed Debugging*, pp. 167–174. ACM Press, 1991.
15. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.
16. P. Fraigniaud, A. Korman, and D. Peleg. Local distributed decision. *FOCS*, pp. 708–717. IEEE, 2011.
17. P. Fraigniaud, S. Rajsbaum, and C. Travers. Locality and checkability in wait-free computing. *Distributed Computing*, 26(4):223–242, 2013.
18. P. Fraigniaud, S. Rajsbaum, and C. Travers. On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems *Technical report #hal-01011079*, 2014. Available at `http://hal.inria.fr/hal-01011079`.
19. A. Genon, T. Massart, C. Meuter. Monitoring distributed controllers: When an efficient LTL algorithm on sequences is needed to model-check traces. *Formal Methods* , lncs #4085, pp. 557–57, 2006.
20. J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. *OOPSLA*, pp. 155–174. ACM, 2009.
21. M. Henle. *A Combinatorial Introduction to Topology*. Dover, 1983.
22. M. Herlihy, D. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann-Elsevier, 2013.
23. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.
24. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, 2001.
25. M. Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.
26. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. *ICSE*, pp. 418–427. IEEE, 2004.
27. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Decentralized runtime analysis of multithreaded applications. *IPDPS*. IEEE, 2006.
28. H. Zhu, M.B. Dwyer, and S. Goddard. Predictable runtime monitoring. In *ECRTS*, pp. 173–183. IEEE 2009.