Distrib. Comput. (2008) 20:415–433
DOI 10.1007/s00446-007-0046-9

The weakest failure detectors to boost obstruction-freedom

Rachid Guerraoui · Michał Kapałka · Petr Kouznetsov

Received: 30 December 2006 / Accepted: 4 October 2007 / Published online: 31 October 2007
© Springer-Verlag 2007

Abstract It is considered good practice in concurrent computing to devise shared object implementations that ensure a minimal *obstruction-free* progress property and delegate the task of boosting liveness to independent generic oracles called *contention managers*. This paper determines necessary and sufficient conditions to implement *wait-free* and *non-blocking* contention managers, i.e., contention managers that ensure *wait-freedom* (resp. *non-blockingness*) of any associated *obstruction-free* object implementation. The necessary conditions hold even when universal objects (like compare-and-swap) or random oracles are available in the implementation of the contention manager. On the other hand, the sufficient conditions assume only basic read/write objects, i.e., registers. We show that failure detector $\Diamond\mathcal{P}$ is the weakest to convert any obstruction-free algorithm into a wait-free one, and Ω^*, a new failure detector which we introduce in this paper, and which is strictly weaker than $\Diamond\mathcal{P}$ but strictly stronger than Ω, is the weakest to convert any obstruction-free algorithm into a non-blocking one. We also address the issue of minimizing the overhead imposed by contention management in low contention scenarios. We propose two *intermittent failure detectors* I_{Ω^*} and $I_{\Diamond\mathcal{P}}$ that are in a precise sense equivalent to, respectively, Ω^* and $\Diamond\mathcal{P}$, but allow for reducing the cost of failure detection in eventually synchronous systems when there is little contention. We present two contention managers: a non-blocking one and a wait-free one, that use, respectively, I_{Ω^*} and $I_{\Diamond\mathcal{P}}$. When there is no contention, the first induces very little overhead whereas the second induces some non-trivial overhead. We show that wait-free contention managers, unlike their non-blocking counterparts, impose an inherent non-trivial overhead even in contention-free executions.

Keywords Shared memory · Obstruction-free · Non-blocking · Wait-free · Contention manager · Failure detector

This paper is a revised and extended version of a paper with the same title in the Proceedings of the 20th International Symposium on Distributed Computing (DISC'06).

R. Guerraoui · M. Kapałka (✉)
School of Computer and Communication Sciences,
EPFL, Lausanne, Switzerland
e-mail: michal.kapalka@epfl.ch

R. Guerraoui
Computer Science and Artificial Intelligence Laboratory,
MIT, Cambridge, USA

P. Kouznetsov
Max Planck Institute for Software Systems,
Saarbrucken, Germany

1 Introduction

Multiprocessor systems are becoming more and more common nowadays. Multithreading thus becomes the norm and studying scalable and efficient synchronization methods is essential. Traditional locking-based techniques do not scale and may induce priority inversion, deadlock and fault-tolerance issues when a large number of threads is involved.

Wait-free synchronization algorithms [18] circumvent the issues of locking and guarantee individual progress even in presence of high contention. Wait-freedom is a liveness property which stipulates that every process completes every operation in a finite number of its own steps, regardless of the status of other processes, i.e., contending or even crashed. Ideal synchronization algorithms combine wait-freedom with *linearizability* [3,21], a safety property which provides the illusion of instantaneous operation executions.

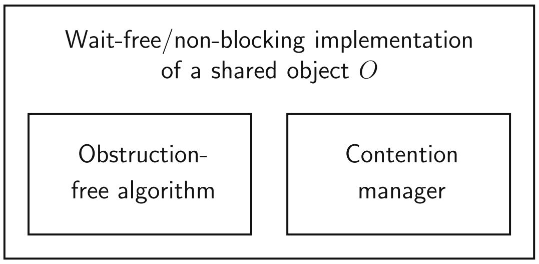
Wait-free/non-blocking implementation of a shared object O
Obstruction-free algorithm
Contention manager

Fig. 1 A modular implementation of a shared object O

Alternatively, a liveness property called *non-blockingness*[1] may be considered instead of wait-freedom. Non-blockingness guarantees global progress, i.e., that some process will complete an operation in a finite number of steps, regardless of the behaviour of other processes. Non-blockingness is weaker than wait-freedom as it might not protect some processes from starvation.

1.1 Obstruction-freedom and contention managers

Wait-free and non-blocking algorithms are, however, notoriously difficult to design [4,23], especially with the practical goal to be fast in low contention scenarios, which are usually considered the most common in practice. An appealing principle to reduce this difficulty consists in separating two concerns of a synchronization algorithm: (1) ensuring linearizability with a weak conditional progress guarantee, and (2) boosting progress. More specifically, the idea is to focus on algorithms that ensure linearizability together with a weak liveness property called *obstruction-freedom* [20], and then combine these algorithms with separate generic oracles that boost progress, called *contention managers* [15,19, 27,28] (see Fig. 1). This separation lies at the heart of modern (obstruction-free) software transactional memory (STM) frameworks [19]. The approach simplifies the task of programmers, for they do not have to care about wait-freedom or non-blockingness. Instead they can focus on safety properties of their implementation, knowing that liveness can be boosted later, using a generic contention manager, possibly developed by concurrency experts and optimized for a given system.

An obstruction-free (or OF, for short) algorithm ensures progress only for processes that execute in isolation for sufficiently long time. In presence of high contention, however, OF algorithms can livelock, preventing any process from terminating. Contention managers are used precisely to cope with high contention scenarios. When queried by a process executing an OF algorithm, a contention manager can delay the process for some time in order to boost the progress of other processes. The contention manager can neither share objects with the OF algorithm, nor return results on its behalf. If it did, the contention manager could peril the safety of the OF algorithm, hampering the overall separation of concerns principle.

In short, the goal of a contention manager is to provide processes with enough time without contention so that they can complete their operations. In its simplest form, a contention manager can be a randomized back-off protocol. More sophisticated contention management strategies have been experimented in practice [16,27,28]. Precisely because they are entirely devoted to progress, they can be combined or changed on the fly [15]. Most previous strategies were *pragmatic*, with no aim to provide *worst case guarantees*. In this paper we focus on contention managers that provide such guarantees. More specifically, we study contention managers that convert any OF algorithm into a non-blocking or wait-free one, and which we call, respectively, *non-blocking* or *wait-free* contention managers.

1.2 Contention management and failure detectors

Two wait-free contention managers have recently been proposed [10,14]. Both rely on timing assumptions to detect processes that fail in the middle of their operations. (The notion of failure might for instance model the fact that a process is swapped-out by the operating system for a long period.) This suggests that *some* information about failures might inherently be needed by any wait-free contention manager. But this is not entirely clear because, in principle, a contention manager could also use randomization to schedule processes, or even powerful synchronization primitives like compare-and-swap, which is known to be *universal*, i.e., able to wait-free implement any other object [18]. In the parlance of [7], we would like to determine whether a *failure detector* is actually needed to implement a contention manager that provides strong liveness guarantees even in the worst case, and if it is, what is the *weakest* one [6]. Besides the theoretical interest, determining the weakest failure detector $\mathscr{D}$ for a given contention manager C is, we believe, of practical relevance, for it provides a uniform implementation of C in any system where $\mathscr{D}$ (and thus C) can actually be implemented.

A failure detector is a distributed oracle that periodically outputs, at each process, some information about which processes are still alive and which have already crashed (failed). Failure detectors differ in the quality of information they provide. For example, *perfect failure detector* $\mathscr{P}$ [7] ensures, intuitively, that every failure is eventually detected by every correct (i.e., non-faulty) process and that there is never any false detection. On the other hand, *eventually perfect failure detector* $\Diamond\mathscr{P}$ [7] gives the same guarantees as $\mathscr{P}$ but

[1] The term *non-blocking* is defined here in the traditional way [18]: "some process will complete its operation in a finite number of steps, regardless of the relative execution speeds of the processes." This term is sometimes confused with the term *lock-free*.

only after some unknown, but finite, time. That is, the output of $\Diamond\mathscr{P}$ can be arbitrary for any finite period of time, but eventually it stabilizes and becomes as accurate as for $\mathscr{P}$. The common property of $\mathscr{P}$ and $\Diamond\mathscr{P}$ is that they provide each alive process with some information about the status of *every* other process. However, this is not always the case. For example, failure detector Ω [6] provides processes with a *leadership* information: it guarantees that eventually all correct processes will elect the same *correct* process as their leader.

Clearly, any output of $\mathscr{P}$ is also a valid output of $\Diamond\mathscr{P}$. Thus, having $\mathscr{P}$ in a system, we can also have $\Diamond\mathscr{P}$ at no cost. It is also straightforward to *implement* Ω in a system that already has $\mathscr{P}$: every process simply chooses the non-crashed (according to $\mathscr{P}$) process with the lowest id as its leader. In fact, if we used $\Diamond\mathscr{P}$ instead of $\mathscr{P}$ in this algorithm, we would still have a correct implementation of Ω. On the contrary, if we only have Ω in a system, we can implement neither $\mathscr{P}$ nor $\Diamond\mathscr{P}$. Also, it is impossible to transform the output of $\Diamond\mathscr{P}$ into a valid output of $\mathscr{P}$ in an asynchronous system.

The above discussion highlights a way in which failure detectors can be compared. Informally, a failure detector $\mathscr{D}$ is said to be *weaker* than a failure detector $\mathscr{D}'$ if we can implement $\mathscr{D}$ using $\mathscr{D}'$ in an asynchronous system [6]. If the opposite is not true, i.e., if we cannot implement $\mathscr{D}'$ out of $\mathscr{D}$, then $\mathscr{D}$ is said to be *strictly weaker* than $\mathscr{D}'$.

1.3 Minimal failure detectors to boost obstruction-freedom

We show in this paper that the eventually perfect failure detector $\Diamond\mathscr{P}$ is the weakest to implement a wait-free contention manager. We also introduce a failure detector Ω^*, which we show is the weakest to implement a non-blocking contention manager. Failure detector Ω^* is strictly weaker than $\Diamond\mathscr{P}$, and strictly stronger than failure detector Ω, known to be the weakest to wait-free implement the (universal) consensus object [6].

It might look surprising that Ω is not sufficient to implement a wait-free or even a non-blocking contention manager. For example, the seminal Paxos algorithm [24] uses Ω to transform an OF implementation of consensus into a wait-free one [5]. Each process that is eventually elected leader by Ω is given enough time to run alone, reach a decision and communicate it to the others. This approach does not help, however, if we want to make sure that processes make progress regardless of the actual (possibly long-lived) object and its OF implementation. Intuitively, the leader elected by Ω may have no operation to perform while other processes may livelock forever. Because a contention manager cannot make processes help each other, the output of Ω is not sufficient: this is so even if randomized oracles or universal objects are available. Intuitively, wait-free contention managers need a failure detector that would take care of *every* non-crashed process with a pending operation so that the process can run alone for sufficiently long time. As for non-blocking contention managers, at least *one* correct process with a pending operation should be given enough time to run alone.

To prove each of the weakest failure detector results, we first present (necessary part) a *reduction* algorithm [6] that *extracts* the output of failure detector Ω^* (respectively, $\Diamond\mathscr{P}$) using a non-blocking (respectively, wait-free) contention manager implementation. When devising our reduction algorithms, we do not restrict what objects (or random oracles) can be used by the contention manager or the OF algorithm. Then (sufficient part) we present algorithms that implement the contention managers using the failure detectors and simple register objects.

It is worthwhile noticing that proving the results goes through defining the notions of non-blocking and wait-free contention managers and specifying the interactions between OF algorithms and contention managers. These, we believe, are interesting contributions in their own right.

1.4 Reducing the cost of contention management

Our implementations of contention managers use failure detectors Ω^* and $\Diamond\mathscr{P}$. In some systems it is possible to implement Ω^* and $\Diamond\mathscr{P}$ efficiently, e.g., when there is some failure detection functionality in the operating system [4]. In general, however, when timeout-based mechanisms have to be used, this is not the case. The problem with failure detectors in their conventional form [7] is that their output cannot depend on computations being performed by processes. Thus, a timeout-based implementation of Ω^* (and a fortiori of $\Diamond\mathscr{P}$) will have to make processes exchange "heartbeat" signals even when failure detection is not actually needed. For example, in executions with low contention between processes, a failure detector might not be necessary at all, and a simple contention manager using, for example, an exponential back-off scheme would be sufficient to provide progress. Ideally we would like a failure detection mechanism to be involved only when needed.

To cope with this issue, we introduce the notion of an *intermittent failure detector* (IFD). Although an IFD is not a failure detector in the sense of [7], it gives processes some information about failures. There are however two important specificities of an IFD. First, its modules, running at different processes, can be stopped and restarted at any time independently. Thus, a process that does not need any information about failures simply stops its local IFD module. Second, intermittent failure detectors return only information about failures a process explicitly queried for, similarly to [9]. This enables frugal IFD implementations.

We present two intermittent failure detectors, I_{Ω^*} and $I_{\Diamond\mathscr{P}}$, and example implementations of theirs in eventually synchronous systems. We establish a formal relationship between a failure detector $\mathscr{D}$ and its intermittent variant $I_{\mathscr{D}}$, by proving that the latter provides as much information about failures as the former. We do so by treating $I_{\mathscr{D}}$ as an abstract problem and proving that $\mathscr{D}$ is *the weakest failure detector* to implement $I_{\mathscr{D}}$. Intuitively, in the worst case (in terms of contention), I_{Ω^*} and $I_{\Diamond\mathscr{P}}$ give processes the same amount of information about failures as, respectively, Ω^* or $\Diamond\mathscr{P}$. However, in many scenarios intermittent failure detectors can be used in a more efficient way than their failure detector counterparts. Namely, a process triggers a failure detection mechanism only when the process needs some information about failures. Clearly, this may cause failures to be detected with a much larger delay than for classical failure detectors. However, in the arguably most common scenarios of low contention and low failure rate, intermittent failure detection is appealing.

We present a non-blocking contention manager CM_{nb} and a wait-free contention manager CM_{wf} that use intermittent failure detectors I_{Ω^*} and $I_{\Diamond\mathscr{P}}$, respectively. Both contention managers can be easily combined with heuristic contention management strategies [27,28] to achieve good average-case performance. Also, both are minimal in terms of failure information. In executions with no contention, i.e., when processes always run alone, contention manager CM_{nb} imposes very little overhead: its implementation (together with the underlying implementation of IFD I_{Ω^*}) does not require any communication between processes. On the contrary, contention manager CM_{wf} provides some level of overhead which we prove is unavoidable. While doing so, we exhibit an interesting "overhead gap" between non-blocking and wait-free contention management.

1.5 Related work

Obstruction-freedom, as a weak liveness property, was introduced by Herlihy et al. [20]. They proposed to delegate stronger progress guarantees to specialized *contention management* oracles. In this paper, we present implementations of contention managers that ensure progress (non-blockingness or wait-freedom) when combined with any obstruction-free algorithm.

Our contention manager implementations share many similarities with the algorithms of [10] and [14], both of which ensure wait-freedom, but use timeout-based failure detection mechanisms directly. In fact, the techniques used in all these algorithms originate in indulgent algorithms [13] designed for partially synchronous systems [7,8,24], ported later to shared memory systems [5,12]. However, the way our contention managers obtain information about failures—from (intermittent) failure detectors—and the way they can be combined with heuristic contention management techniques are, we believe, novel. The implementations of IFDs I_{Ω^*} and $I_{\Diamond\mathscr{P}}$ we give in this paper are similar to known message passing implementations of $\Diamond\mathscr{P}$ [1,7,9,25].

1.6 Roadmap

The paper is organized as follows. Section 2 presents our system model and formally defines wait-free and non-blocking contention managers. In Sects. 3 and 4, we prove our weakest failure detector results. Then, in Sect. 5, we introduce the abstraction of an intermittent failure detector (IFD) and define IFDs I_{Ω^*} and $I_{\Diamond\mathscr{P}}$. Next, we present the implementations of contention managers CM_{nb} and CM_{wf} (Sect. 6) and example implementations of I_{Ω^*} and $I_{\Diamond\mathscr{P}}$ in eventually synchronous systems (Sect. 7). In Sect. 8, we discuss the overhead of non-blocking and wait-free contention managers. We conclude the paper with some final remarks in Sect. 9.

2 Preliminaries

2.1 Processes and failure detectors

We consider a set of n processes $\Pi = \{p_1, \ldots, p_n\}$ in a shared memory system [18,22]. A process executes the (possibly randomized) algorithm assigned to it, until the process *crashes* (*fails*) and stops executing any action. We assume the existence of a global discrete clock that is, however, inaccessible to the processes. We say that a process is *correct* if it never crashes. We say that process p_i is *alive* at time t if p_i has not crashed by time t.

A *failure detector* [6,7] is a distributed oracle that provides every process with some information about failures. The output of a failure detector depends only on which and when processes fail, and not on computations being performed by the processes. A process p_i queries a failure detector $\mathscr{D}$ by accessing local variable $\mathscr{D}$-*output*$_i$—the output of the module of $\mathscr{D}$ at process p_i. Failure detectors can be partially ordered according to the amount of information about failures they provide. A failure detector $\mathscr{D}$ is *weaker than a failure detector* $\mathscr{D}'$, and we write $\mathscr{D} \preceq \mathscr{D}'$, if there is an algorithm (called a *reduction* algorithm) that uses $\mathscr{D}'$ (as the only source of information about failures) to emulate the output of $\mathscr{D}$ [6]. If $\mathscr{D} \preceq \mathscr{D}'$ but $\mathscr{D}' \not\preceq \mathscr{D}$, we say that $\mathscr{D}$ *is strictly weaker than* $\mathscr{D}'$, and we write $\mathscr{D} \prec \mathscr{D}'$.

2.2 Base and high-level objects

Processes communicate by invoking primitive operations (which we will call *instructions*) on *base* shared objects and seek to implement the *operations* of a *high-level* shared object O. Object O is in turn used by an application, as a high-level inter-process communication mechanism. We call invocation and response events of a high-level operation *op* on the implemented object O *application events* and denote them by, respectively, $inv(op)$ and $ret(op)$ (or $inv_i(op)$ and $ret_i(op)$ at a process p_i).

An *implementation* of O is a distributed algorithm that specifies, for every process p_i and every operation op of O, the sequences of *steps* that p_i should take in order to complete op. Process p_i *completes* operation op when p_i returns from op. Every process p_i may complete any number of operations but, at any point in time, at most one operation op can be *pending* (started and not yet completed) at p_i.

We consider implementations of O that combine a sub-protocol that ensures safety and a weak liveness property, called *obstruction-freedom*, with a sub-protocol that boosts this liveness guarantee. The former is called an *obstruction-free (OF)* algorithm A and the latter a *contention manager* CM. We focus on *linearizable* [3,21] implementations of O: every operation appears to the application as if it took effect instantaneously between its invocation and its return. An implementation of O involves two categories of steps executed by any process p_i: those (executed on behalf) of CM and those (executed on behalf) of A. In each step, a process p_i either executes an instruction on a base shared object or queries a failure detector. The latter case occurs only if p_i executes a step on behalf of CM.

Obstruction-freedom [19,20] stipulates that if a process invokes an operation op on object O and from some point in time executes steps of A alone,[2] then the process eventually completes op. *Non-blockingness* stipulates that if some correct process never completes an invoked operation, then some other process completes infinitely many operations. *Wait-freedom* [18] is stronger and ensures that every correct process that invokes an operation eventually returns from the operation.

2.3 Interaction between modules

OF algorithm A, executed by any process p_i, communicates with contention manager CM via *calls* try_i and $resign_i$ implemented by CM (see Fig. 2). Process p_i invokes try_i just after p_i starts an operation, and also later (even several times before p_i completes the operation) to signal possible contention. Process p_i invokes $resign_i$ just before returning from an operation, and always eventually returns from this call (or crashes). Both calls, try_i and $resign_i$, return *ok*.

An example OF algorithm that uses this model of interaction with a contention manager is Algorithm 1. The algorithm implements a timestamping mechanism and is based on the implementation of a splitter [26]. It is not meant to be practical or efficient—it just shows how calls *try* and *resign* should be used.

[2] That is without encountering *step contention* [2].

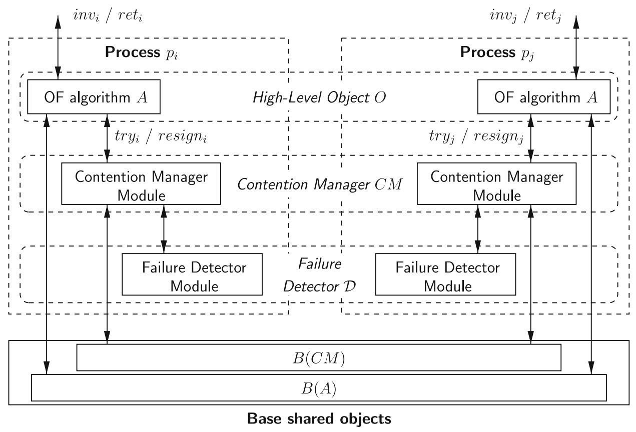

Fig. 2 The OF algorithm/contention manager interface

Algorithm 1: An example OF algorithm implementing a timestamping mechanism (code for process p_i)

```
uses: $A[1,\ldots]$—unbounded array of registers,
      $B[1,\ldots]$—unbounded array of single-bit registers, $L$—a
      register
initially: $A[1,\ldots] \leftarrow \bot$, $B[1,\ldots] \leftarrow false$, $L \leftarrow 1$
1.1  upon of-getTimestamp do
1.2      $CM.try_i$
1.3      $j \leftarrow L$
1.4      while true do
1.5          $A[j] \leftarrow i$
1.6          if $B[j] = false$ then
1.7              $B[j] \leftarrow true$
1.8              if $A[j] = i$ then
1.9                  $L \leftarrow j + 1$
1.10                 $CM.resign_i$
1.11                 return $j$
1.12         $CM.try_i$
1.13         $j \leftarrow j + 1$
```

The intuition behind the algorithm is the following. A process p_i that invokes *of-getTimestamp* scans the array B of registers, starting from the index stored in register L, to find the lowest value j for which $B[j]$ is *false*. Then, p_i sets $B[j]$ to *true*. (The value of $B[j]$ will never change thereafter.) If no other process is executing steps concurrently to p_i, then p_i returns j as a new timestamp. Also, p_i stores value $j+1$ in register L to optimize future invocations of *of-getTimestamp*. The splitter code in lines 1.5–1.8 guarantees that *at most one* process can return a given value j, thus ensuring that the timestamps are unique. However, if two or more processes execute the algorithm concurrently, it might happen that none of them ever returns. That is why it is important that a contention manager delays some processes and let only one execute steps of the algorithm at a time.

We denote by $B(A)$ and $B(CM)$ the sets of base shared objects, always *disjoint*, that can be possibly accessed by steps of, respectively, A and CM, in every execution, by every process. Calls *try* and *resign* are thus the only means by which A and CM interact. The events corresponding to invocations of, and responses from, *try* and *resign* are called *cm-events*. We denote by try_i^{inv} and $resign_i^{\text{inv}}$ an invocation of call try_i and $resign_i$, respectively (at process p_i), and by try_i^{ret} and $resign_i^{\text{ret}}$—the corresponding responses.

2.4 Executions and histories

An *execution* of an OF algorithm A combined with a contention manager CM is a sequence of *events* that include steps of A, steps of CM, cm-events and application events. Every event in an execution is associated with a unique time representing the moment at which the event took place. Simultaneous events (say in case of multiprocessors) are arbitrarily ordered. Every execution e induces a *history* $H(e)$ that includes only application events (invocations and responses of high-level operations). The corresponding *CM-history* $H_{\text{CM}}(e)$ is the longest subsequence of e containing only application events and cm-events of the execution, and the corresponding *OF-history* $H_{\text{OF}}(e)$ is the longest subsequence of e containing only application events, cm-events, and steps of A. For a sequence s of events, $s|i$ denotes the longest subsequence of s containing only events at process p_i.

We say that a process p_i is *blocked* at time t in an execution e if (1) p_i is alive at time t, and (2) the latest event in $H_{\text{CM}}(e)|i$ that occurred before t is try_i^{inv} or $resign_i^{\text{inv}}$. A process p_i is *busy* at time t in e if (1) p_i is alive at time t, and (2) the latest event in $H_{\text{CM}}(e)|i$ that occurred before t is try_i^{ret}. We say that a process p_i is *active* at t in e if p_i is either busy or blocked at time t in e. We say that a process p_i is *idle* at time t in e if p_i is not active at t in e.[3] A process *resigns* when it invokes *resign* on a contention manager.

We say that a process p_i is *obstruction-free* in an interval $[t, t']$ in an execution e, if p_i is the only process that takes steps of A in $[t, t']$ in e and p_i is not blocked infinitely long in $[t, t']$ (if $t' = \infty$). We say that process p_i is *eventually obstruction-free* at time t in e if p_i is active at t or later and p_i either resigns after t or is obstruction-free in the interval $[t', \infty)$ for some $t' > t$. Note that, since algorithm A is obstruction-free, if a correct active process p_i is eventually obstruction-free at some point in time, then p_i eventually resigns and completes its operation thereafter.

2.5 Well-formed executions

We impose certain restrictions on the way an OF algorithm A and a contention manager CM interact. In particular, we assume that no process takes steps of A while being blocked by CM or idle, and no process takes infinitely many steps of A without calling CM infinitely many times. Further, a process must inform CM that an operation is completed by calling *resign* before returning the response to the application.

Formally, we assume that every execution e is *well-formed*, i.e., $H(e)$ is linearizable [3,21], and, for every process p_i, (1) $H_{\text{CM}}(e)|i$ is a prefix of a sequence $[op_1][op_2], \ldots,$ where each $[op_k]$ has the form $inv_i(op_k)$, try_i^{inv}, $try_i^{\text{ret}}, \ldots, try_i^{\text{inv}}$, try_i^{ret}, $resign_i^{\text{inv}}$, $resign_i^{\text{ret}}$, $ret_i(op_k)$; (2) in $H_{\text{OF}}(e)|i$, no step of A is executed when p_i is blocked or idle, (3) in $H_{\text{OF}}(e)|i$, inv_i can only be followed by try_i^{inv}, and ret_i can only be preceded by $resign_i^{\text{ret}}$; (4) if p_i is busy at time t in e, then at some $t' > t$, process p_i is idle or blocked. The last condition implies that every busy process p_i eventually invokes try_i (and becomes blocked), resigns or crashes. Clearly, in a well-formed execution, every process goes through the following cyclical order of modes: *idle, active, idle, ...,* where each *active* period consists itself of a sequence *blocked, busy, blocked, ...*.

2.6 Non-blocking contention manager

We say that a contention manager *CM guarantees non-blockingness for an OF algorithm A* if in each execution e of A combined with CM the following property is satisfied: if some correct process is active at a time t, then at some time $t' > t$ some process resigns.

We say that a contention manager CM is *non-blocking* if, for every OF algorithm A, in every execution of A combined with CM the following property is ensured at every time t:

Global Progress. If some correct process is active at t, then some correct process is eventually obstruction-free at t.

Intuitively, a non-blocking contention manager allows at least one active process to be obstruction-free (and busy) for sufficiently long time, so that the process can complete its operation.

Theorem 1 *A contention manager CM guarantees non-blockingness for every OF algorithm if and only if CM is non-blocking.*

Proof (⇒) Consider a contention manager CM that guarantees non-blockingness for every OF algorithm. Let A be any OF algorithm and e be any execution of A combined with CM. Let some correct process be active at time t in e. Since CM guarantees non-blockingness, some active process resigns at some future time, and the Global Progress property is trivially ensured.

(⇐) By contradiction, assume that there exists a non-blocking contention manager CM such that, for some OF algorithm A, there is an execution e of A combined with CM, such that some correct process is active at t, and no active process resigns after t. By Global Progress, some correct active process p_i is eventually obstruction-free at t. Since

[3] Note that every process that has crashed is permanently idle.

A is obstruction-free and p_i takes infinitely many steps of A in isolation, p_i must complete its operation and resign after t—a contradiction. □

2.7 Wait-free contention manager

We say that a contention manager *CM guarantees wait-freedom for an OF algorithm A* if in every execution e of A combined with *CM* the following property is satisfied: if a process p_i is active at a time t, then at some time $t' > t$, p_i becomes idle. In other words, every operation executed by a correct process eventually returns.

A contention manager *CM* is *wait-free* if, for every OF algorithm A, in every execution of A combined with *CM*, the following property is ensured at every time t:[4]

Fairness. If a correct process p_i is active at t, then p_i is eventually obstruction-free at t.

Intuitively, a wait-free contention manager makes sure that every correct active process is given "enough" time to complete its operation, regardless of how other processes behave.

Theorem 2 *A contention manager CM guarantees wait-freedom for every OF algorithm if and only if CM is wait-free.*

Proof (⇒) Consider a contention manager *CM* that guarantees wait-freedom for every OF algorithm. Let A be any OF algorithm and e be any execution of A combined with *CM*. Since in e every active process is eventually idle, every correct active process eventually resigns in e, and so the Fairness property is trivially satisfied.

(⇐) Let *CM* be a wait-free contention manager, and A be any OF algorithm. Consider any execution e of A combined with *CM*.

Suppose, by contradiction, that some correct process p_i is active at time t and never completes its operation thereafter. But then, by Fairness, p_i is eventually obstruction-free at t and so p_i is obstruction-free in period $[t', \infty)$ for some $t' > t$. Therefore, since A is obstruction-free and p_i takes infinitely many steps of A in isolation, p_i must eventually resign and complete its operation—a contradiction. □

In the following, we seek to determine the *weakest* [6] failure detector $\mathscr{D}$ to implement a non-blocking (resp. wait-free) contention manager *CM*. This means that (1) $\mathscr{D}$ implements such a contention manager, i.e., there is an algorithm that implements *CM* using $\mathscr{D}$, and (2) $\mathscr{D}$ is *necessary* to implement such a contention manager, i.e., if a failure detector $\mathscr{D}'$ implements *CM*, then $\mathscr{D} \preceq \mathscr{D}'$. In our context, a reduction algorithm that transforms $\mathscr{D}'$ into $\mathscr{D}$ uses the $\mathscr{D}'$-based implementation of the corresponding contention manager as a "black box" and read-write registers.

[4] This property is ensured by wait-free contention managers from the literature [10,14].

3 Non-blocking contention managers

3.1 Failure detector Ω^*

Let $S \subseteq \Pi$ be a non-empty set of processes. Failure detector Ω_S outputs, at every process, an identifier of a process (called a *leader*), such that all correct processes *in S* eventually agree on the identifier of the same *correct* process *in S*.[5]

Failure detector Ω^* is the composition $\{\Omega_S\}_{S \subseteq \Pi, S \neq \emptyset}$: at every process p_i, Ω^*-*output*$_i$ is a tuple consisting of the outputs of failure detectors Ω_S. We position Ω^* in the hierarchy of failure detectors of [7] by proving the following theorem:

Theorem 3 $\Omega \prec \Omega^* \prec \Diamond\mathscr{P}$.

Proof It is immediate that Ω is weaker than Ω^*: Ω_Π is the same as Ω. In a system of three or more processes, Ω is strictly weaker than Ω^*. Indeed, consider a system of three processes, p_1, p_2, and p_3, and assume, by contradiction, that Ω^* is weaker than Ω, i.e., that there exists a reduction algorithm $T_{\Omega \to \Omega^*}$ which extracts the output of Ω^* using Ω. Take an execution e of $T_{\Omega \to \Omega^*}$ in which p_3 is correct, p_2 is faulty, Ω always outputs p_3 at every process and consider the emulated output of $\Omega_{\{p_1,p_2\}}$. Since p_1 is the only correct process in $\{p_1, p_2\}$, there is a finite prefix e' of e in which $\Omega_{\{p_1,p_2\}}$ outputs p_1 at p_1. But this finite execution is indistinguishable from a finite execution e'' in which p_2 is correct but slow. Now consider a finite extension of e'' in which p_1 fails, and thus eventually $\Omega_{\{p_1,p_2\}}$ outputs p_2 at p_2. But this finite execution is indistinguishable from a finite execution in which p_1 is correct but slow. By repeating this argument, we obtain an infinite execution of $T_{\Omega \to \Omega^*}$ in which both p_1 and p_2 are correct, and the output $\Omega_{\{p_1,p_2\}}$ never stabilizes at a single correct process—a contradiction.

It is immediate that Ω^* is weaker than $\Diamond\mathscr{P}$: eventually each correct process p_i has complete and accurate information from $\Diamond\mathscr{P}$ about failures of all other processes, so p_i can perform an eventually perfect leader election in each subset of processes p_i belongs to, thus extracting the output of Ω^*.

To show that Ω^* is *strictly* weaker than $\Diamond\mathscr{P}$, consider a system of two processes, p_1 and p_2, and assume, by contradiction, that $\Diamond\mathscr{P}$ is weaker than Ω^*, i.e., that there exists a reduction algorithm $T_{\Omega^* \to \Diamond\mathscr{P}}$ which extracts the output of $\Diamond\mathscr{P}$ using Ω^*.

[5] Ω_S can be seen as a restriction of the eventual leader election failure detector Ω [6] to processes in S. The definition of Ω_S resembles the notion of Γ-accurate failure detectors introduced in [17]. Clearly, Ω_Π is Ω.

Using $T_{\Omega^* \to \Diamond\mathscr{P}}$, we implement $\Diamond\mathscr{P}$ in an *asynchronous system*, establishing a contradiction with [7,11]. In the implementation, the processes run two parallel algorithms, T_1 and T_2. The algorithm T_i $(i = 1, 2)$ is identical to $T_{\Omega^* \to \Diamond\mathscr{P}}$, except that, instead of querying Ω^*, it assumes that the $\Omega_{\{p_1,p_2\}}$ component of Ω^* always outputs p_i at each process. (Clearly, $\Omega_{\{p_i\}}$ must always output p_i.) Note that every finite execution of T_i is also a finite execution of $T_{\Omega^* \to \Diamond\mathscr{P}}$. If p_i is correct, then every (even infinite) execution of T_i is also an execution of $T_{\Omega^* \to \Diamond\mathscr{P}}$. Thus, every process p_i (correct or not) obtains from T_i a valid output of $\Diamond\mathscr{P}$. But T_i does not use any failure detector, and so we get an implementation of $\Diamond\mathscr{P}$ in an asynchronous system. □

3.2 The necessity part

To show that Ω^* is necessary to implement a non-blocking contention manager, it suffices to prove that, for every non-empty $S \subseteq \Pi$, Ω_S is necessary to implement a non-blocking contention manager. Let *CM* be a non-blocking contention manager using failure detector $\mathscr{D}$. We show that $\Omega^* \preceq \mathscr{D}$ by presenting an algorithm $T_{\mathscr{D} \to \Omega_S}$ (Algorithm 2) that, using *CM* and $\mathscr{D}$, emulates the output of Ω_S.

Algorithm 2: Extracting Ω_S from a non-blocking contention manager (code for each processes p_i from set S; others are permanently idle)

```
  uses: $L$—register
  initially: $\Omega_S$-$output_i \leftarrow p_i$, $L \leftarrow$ some process in $S$
    Launch two parallel tasks: $T_i$ and $F_i$
2.1 parallel task $F_i$
2.2     $\Omega_S$-$output_i \leftarrow L$
2.3 parallel task $T_i$
2.4     while true do
2.5         issue $try_i$ and wait until busy (i.e., until call $try_i$ returns)
2.6         $L \leftarrow p_i$  // announce yourself a leader
```

The algorithm works as follows. Every process $p_i \in S$ runs two parallel tasks T_i and F_i. In task T_i, process p_i periodically (1) gets blocked by *CM* after invoking try_i (line 2.5), and (2) once p_i gets busy again, announces itself a leader for set S by writing its id in L (line 2.6). In task F_i, process p_i periodically determines its leader by reading register L (line 2.2).[6]

Thus, no process ever resigns and every correct process in S is permanently active from some point in time. Intuitively, this signals a possible livelock to *CM* which has to eventually block all active processes except for one that should run obstruction-free for sufficiently long time. By Global Progress, *CM* cannot block *all* active processes forever, and so if the elected process crashes (and so becomes idle), *CM* lets another active process run obstruction-free. Eventually, all correct processes in S agree on the same correct process in S. Processes outside S are permanently idle and permanently output their own ids: they do not access *CM*.

This approach contains a subtlety. To make sure that there is a time after which the same correct leader in S is permanently elected by the correct processes in S, we do not allow the elected leader to resign (the output of Ω_S has to be eventually stable). This violates the assumption that processes using *CM* run an obstruction-free algorithm, and thus, a priori, *CM* is not obliged to preserve Global Progress. However, as we show below, since *CM* does not "know" how much time a process executing an OF algorithm requires to complete its operation, *CM* has to provide some correct process with *unbounded* time to run in isolation.

Theorem 4 *Every non-blocking contention manager can be used to implement failure detector Ω^*.*

Proof Let $S \subseteq \Pi$, $S \neq \emptyset$ and consider any execution of Algorithm 2. If S contains no correct process, then Ω_S-$output_i$ (for every process $p_i \in S$) trivially satisfies the property of Ω_S. Now assume that there is a correct process in S. We claim that *CM* eventually lets exactly one correct process in S run obstruction-free while blocking forever all the other processes in S.

Suppose not. We obtain an execution in which every correct process in S is allowed to be obstruction-free only for bounded periods of time. But the CM-history of this execution corresponds to an execution of some OF algorithm A combined with *CM* in which no active process ever completes its operation because no active process ever obtains enough time to run in isolation. Thus, no active process is eventually obstruction-free in that execution. This contradicts the assumption that *CM* is non-blocking.

Therefore, there is a time after which exactly one correct process $p_j \in S$ is periodically busy (others are blocked or idle forever) and, respectively, register L permanently stores the identifier of p_j. Thus, eventually, every correct process in S outputs p_j: the output of Ω_S is extracted. □

3.3 The sufficiency part

We describe an implementation of a non-blocking contention manager using Ω^* and registers in Algorithm 3. The algorithm works as follows. All active processes, upon calling *try*, participate in the leader election mechanism using Ω^* in lines 3.3–3.5. The active process p_i that is elected a leader returns from *try* and is (eventually) allowed to run

[6] If a process is blocked in one task, it continues executing steps in parallel tasks.

obstruction-free until p_i resigns. Once p_i resigns, the processes elect another leader. Failure detector Ω^* guarantees that if an active process is elected and crashes before resigning, another active process is eventually elected.

Algorithm 3: A non-blocking contention manager using $\Omega^* = \{\Omega_S\}_{S \subseteq \Pi, S \neq \emptyset}$ (code for process p_i)

```
uses: $T[1, \ldots, n]$—array of single-bit registers
initially: $T[1, \ldots, n] \leftarrow$ false

3.1 upon $try_i$ do
3.2     $T[i] \leftarrow$ true
3.3     repeat
3.4         $S \leftarrow \{p_j \in \Pi \mid T[j] = true\}$
3.5     until $\Omega_S$-$output_i = p_i$

3.6 upon $resign_i$ do
3.7     $T[i] \leftarrow$ false
```

Lemma 1 *The contention manager implemented by Algorithm 3 guarantees non-blockingness for every OF algorithm.*

Proof Assume, by contradiction that there exists an OF algorithm A for which contention manager CM implemented by Algorithm 3 does not guarantee non-blockingness, i.e., there exists an execution e of A combined with CM in which there are a correct process p_i and a time t, such that p_i is active at t but for all $t' > t$, no active process resigns at t'.

Take any time $t' > t$. Let us denote by $S(t')$ the set of all processes p_j such that $T[j] = true$ at time t' in e. Since no active process resigns after t, there is a time $t^* \geq t$ and a set S, such that for all $t' > t^*$, $S(t') = S$. By the algorithm, p_i eventually sets $T[j]$ to *true*. Thus, p_i is in S, i.e., S includes at least one correct process. At every correct process in S, Ω_S eventually outputs the same correct process p_j in set S (a leader).

Since every active process eventually invokes *try*, resigns or crashes (by the properties of OF algorithms), and no process resigns after t^*, there is a time $t' > t^*$ after which every correct process except for p_j gets permanently blocked in lines 3.3–3.5. That is because p_j does not resign after t and so p_j does not reset $T[j]$ to *false* thereafter and remains the leader for set S forever. Thus, p_j is eventually obstruction-free at t. Since p_j runs an obstruction-free algorithm A, it eventually resigns and completes its operation—a contradiction. □

From Theorem 1 and Lemma 1 we immediately obtain the following result:

Theorem 5 *Algorithm 3 implements a non-blocking contention manager.*

4 Wait-free contention managers

We prove here that the weakest failure detector to implement a wait-free contention manager is $\Diamond\mathcal{P}$ [7]. Failure detector $\Diamond\mathcal{P}$ outputs, at each time and every process, a set of *suspected* processes. There is a time after which (1) every crashed process is permanently suspected by every correct process and (2) no correct process is ever suspected by any correct process.

4.1 The necessity part

We first consider a wait-free contention manager CM using a failure detector $\mathcal{D}$, and we exhibit a reduction algorithm $T_{\mathcal{D}\rightarrow\Diamond\mathcal{P}}$ (Algorithm 4) that, using CM and $\mathcal{D}$, emulates the output of $\Diamond\mathcal{P}$.

Algorithm 4: Extracting $\Diamond\mathcal{P}$ from a wait-free contention manager (code for process p_i)

```
uses: $R[1, \ldots, n]$—array of registers
initially: $\Diamond\mathcal{P}$-$output_i \leftarrow \Pi - \{p_i\}$, $k \leftarrow 0$, $R[i] \leftarrow 0$
  Launch $n(n-1)$ parallel instances of $CM$: $C_{jk}$,
  $j, k \in \{1, \ldots, n\}$, $j \neq k$
  Launch $2n - 1$ parallel tasks: $T_{ij}$, $T_{ji}$, $j \in \{1, \ldots, n\}$, $i \neq j$,
  and $F_i$

4.1  parallel task $F_i$
         // "heartbeat" signal
4.2      while true do $R[i] \leftarrow R[i] + 1$

4.3  parallel task $T_{ij}$, $j = 1, \ldots, i-1, i+1, \ldots, n$
4.4      while true do
4.5          $x_j \leftarrow R[j]$
             // stop suspecting $p_j$
4.6          $\Diamond\mathcal{P}$-$output_i \leftarrow \Diamond\mathcal{P}$-$output_i - \{p_j\}$
4.7          issue $try_i^{ij}$ (in $C_{ij}$) and wait until busy
4.8          issue $resign_i^{ij}$ (in $C_{ij}$) and wait until idle
             // start suspecting $p_j$
4.9          $\Diamond\mathcal{P}$-$output_i \leftarrow \Diamond\mathcal{P}$-$output_i \cup \{p_j\}$
             // wait until $p_j$ takes a new step
4.10         wait until $R[j] > x_j$

4.11 parallel task $T_{ji}$, $j = 1, \ldots, i-1, i+1, \ldots, n$
4.12     while true do issue $try_i^{ji}$ (in $C_{ji}$) and wait until busy
```

We run several instances of CM. These instances use disjoint sets of base shared objects and do not directly interact. Basically, in each instance, only two processes are active and all other processes are idle. One of the two processes, say p_j, gets active and never resigns thereafter, while the other, say p_i, permanently alternates between being active and idle. To CM it looks like p_j is always obstructed by p_i. Thus, to guarantee wait-freedom, the instance of CM has to eventually block p_i and let p_j run obstruction-free until p_j resigns *or crashes*. Therefore, when p_i is blocked, p_i can assume that

p_j is alive and when p_i is busy, p_i can suspect p_j of having crashed, until p_i eventually observes p_j's "heartbeat" signal, which p_j periodically broadcasts using a register. This ensures the properties of $\Diamond\mathcal{P}$ at process p_i, provided that p_j never resigns.

As in Sect. 3, we face the following issue. If p_j is correct, p_i will be eventually blocked forever and p_j will thus be eventually obstruction-free. Hence, in the corresponding execution, obstruction-freedom is violated, i.e., the execution cannot be produced by any OF algorithm combined with *CM*. One might argue then that *CM* is not obliged to preserve Fairness with respect to p_j. However, we show below that, since *CM* does not "know" how much time a process executing an OF algorithm requires to complete its operation, *CM* has to provide p_j with *unbounded* time to run in isolation.

More precisely, the processes in Algorithm 4 run $n(n-1)$ parallel instances of *CM*, denoted each CM_{jk}, where $j, k \in \{1, \ldots, n\}$, $j \neq k$. We denote the events that process p_i issues in instance CM_{jk} by try_i^{jk} and $resign_i^{jk}$. Besides, every process p_i runs $2n-1$ parallel tasks: T_{ij}, T_{ji}, where $j \in \{1, \ldots, n\}$, $i \neq j$, and F_i. Every task T_{ij} executed by p_i is responsible for detecting failures of process p_j. Every task T_{ji} executed by p_i is responsible for preventing p_j from falsely suspecting p_i. In task F_i, p_i periodically writes ever-increasing "heartbeat" values in a shared register $R[i]$.

In every instance CM_{ij}, there can be only two active processes: p_i and p_j. Process p_i cyclically gets active (line 4.7) and resigns (line 4.8), and process p_j gets active once and keeps getting blocked (line 4.12). Each time before p_i gets active, p_i removes p_j from the list of suspected processes (line 4.6). Each time p_i stops being blocked, p_i starts suspecting p_j (line 4.9) and waits until p_i observes a "new" step of p_j (line 4.10). Once such a step of p_j is observed, p_i stops suspecting p_j and gets active again.

Theorem 6 *Every wait-free contention manager can be used to implement failure detector $\Diamond\mathcal{P}$.*

Proof Consider any execution e of $T_{\mathcal{D}\rightarrow\Diamond\mathcal{P}}$, and let p_i be any correct process. We show that, in e, $\Diamond\mathcal{P}$*-output*$_i$ satisfies the properties of $\Diamond\mathcal{P}$, i.e., p_i eventually permanently suspects every non-correct process and stops suspecting every correct process. (Note that if a process p_i is not correct, then $\Diamond\mathcal{P}$*-output*$_i$ trivially satisfies the properties of $\Diamond\mathcal{P}$.)

Let p_j be any process distinct from p_i. Assume p_j is not correct. Thus p_i is the only correct active process in instance CM_{ij}. By the Fairness property of *CM*, p_i is eventually obstruction-free every time p_i becomes active, and so p_i cannot be blocked infinitely long in line 4.7. Since there is a time after which p_j stops taking steps, eventually p_i starts suspecting p_j (line 4.9) and suspends in line 4.10, waiting until p_j takes a new step. Thus, p_i eventually suspects p_j forever.

Assume now that p_j is correct. We claim that p_i must eventually get permanently blocked so that p_j would run obstruction-free from some point in time forever. Suppose not. Then we obtain an execution in which p_i alternates between active and idle modes infinitely many times, and p_j stays active and runs obstruction-free only for bounded periods of time. But the CM-history of this execution could be produced by an execution e' of some OF algorithm combined with *CM* in which p_j never completes its operation because p_j never runs long enough in isolation. Thus, Fairness is violated in execution e' and this contradicts the assumption that *CM* is wait-free. Hence, eventually p_i gets permanently blocked in line 4.7. Since each time p_i is about to get blocked, p_i stops suspecting p_j in line 4.6, there is a time after which p_i never suspects p_j.

Thus, there is a time after which, if p_j is correct, then p_j stops being suspected by every correct process, and if p_j is non-correct, then every correct process permanently suspects p_j. □

4.2 The sufficiency part

We describe an implementation of a wait-free contention manager using $\Diamond\mathcal{P}$ and registers in Algorithm 5. The algorithm relies on a (wait-free) primitive *GetTimestamp()* that generates unique, locally increasing timestamps and makes sure that if a process gets a timestamp *ts*, then no process can get timestamps lower than *ts* infinitely many times (this primitive can be implemented in an asynchronous system using read-write registers). The idea of the algorithm is the following. Every process p_i that gets active receives a timestamp in line 5.2 and announces the timestamp in register $T[i]$. Every active process that invokes *try* repeatedly runs a leader election mechanism (lines 5.3–5.6): the non-suspected (by $\Diamond\mathcal{P}$) process that announced the lowest (non-$\perp$) timestamp is elected a leader. If a process p_i is elected, p_i returns from try_i and becomes busy. $\Diamond\mathcal{P}$ guarantees that eventually the same correct active process is elected by all active processes. All other active processes stay blocked until the process resigns and resets its timestamp in line 5.8. The leader executes steps obstruction-free then. Since the leader runs an OF algorithm, the leader eventually resigns and resets its timestamp in line 5.8, so that another active process, which now has the lowest timestamp in T, can become a leader.

Lemma 2 *The contention manager implemented by Algorithm 5 guarantees wait-freedom for all OF algorithms.*

Proof Consider an execution e of any OF algorithm A combined with contention manager *CM* implemented by Algorithm 5. By contradiction, assume that in e some correct process is active at some time t and never resigns after t. Let V denote the non-empty set of correct processes that

Algorithm 5: A wait-free contention manager using $\Diamond\mathcal{P}$ (code for process p_i)

```
uses: $T[1,\ldots,n]$—array of registers (other variables are local)
initially: $T[1,\ldots,n] \leftarrow \bot$
5.1 upon $try_i$ do
5.2     if $T[i] = \bot$ then $T[i] \leftarrow$ GetTimestamp()
5.3     repeat
5.4         $sact_i \leftarrow \{ j \mid T[j] \neq \bot \wedge p_j \notin \Diamond\mathcal{P}\text{-}output_i \}$
5.5         $leader_i \leftarrow \operatorname{argmin}_{j \in sact_i} T[j]$
5.6     until $leader_i = i$
5.7 upon $resign_i$ do
5.8     $T[i] \leftarrow \bot$
```

are active at some time t but never resign (in line 5.8) and complete their operations thereafter, i.e., that remain active after t forever. Recall that every process in V either invokes *try* infinitely many times or invokes try and stays blocked forever (by the properties of OF algorithms). Let t^* be time after which no process in V resigns, and at which, for every process $p_i \in V$, $T[i] \neq \bot$. Let ts_j^* denote the value of $T[j]$ at time t^*. Since, for every process $p_i \in V$, $ts_i^* \neq \bot$, and no process in V resigns after time t^*, $T[i] = ts_i^*$ at all times $t \geq t^*$.

Let p_i be the process in V having the lowest timestamp in $\{ ts_k^* \mid p_k \in V \}$ (there is exactly one such process since timestamps are unique). We establish a contradiction by showing that p_i has to eventually resign.

Let us consider time $t' > t^*$ after which:

- at every correct process, failure detector $\Diamond\mathcal{P}$ permanently outputs the list of all non-correct processes (by the properties of $\Diamond\mathcal{P}$, this eventually happens),
- all non-correct processes have crashed,
- for every correct process $p_j \neq p_i$, if $T[j] \neq \bot$, then $T[j] > ts_i^*$.

The last condition eventually holds, because timestamps are unique, no process can receive a timestamp lower that ts_i^* infinitely many times and p_i has the lowest timestamp among processes in V (that retain their timestamps infinitely long).

Thus, after t', p_i is always elected a leader, and every correct process p_j other than p_i that gets blocked after time t' will remain blocked in lines 5.3–5.6, as long as p_i does not resign.

Hence, eventually p_i will be the only active process that is not blocked, and thus p_i will be given unbounded time to perform steps of A in isolation. Since A is obstruction-free, p_i eventually resigns and completes its operation—a contradiction. □

From Theorem 2 and Lemma 2 we immediately obtain the following result:

Theorem 7 *Algorithm 5 implements a wait-free contention manager.*

5 Intermittent failure detectors

As discussed Sect. 1, contention manager implementations based on failure detectors might be considered not very effective, especially if the goal is to reduce the complexity of executions with low contention. To cope with this issue, we revisit the notion of failure detectors and introduce here an *intermittent* variant of this notion.

More specifically, we introduce two intermittent failure detectors (IFDs), which can be viewed as intermittent variants of Ω^* and $\Diamond\mathcal{P}$. We denote them by, respectively, I_{Ω^*} and $I_{\Diamond\mathcal{P}}$. Both I_{Ω^*} and $I_{\Diamond\mathcal{P}}$ implement two procedures that can be invoked by a contention manager: *stop* and *query*. The former stops the IFD implementation on the calling process. The latter one restarts the IFD, if it has been stopped, and queries the IFD. We assume that an IFD module at each process is, by default, stopped until the process queries the IFD for the first time.

Intuitively, I_{Ω^*} implements an eventual leader election mechanism among a set $S \subseteq \Pi$ of processes (given as an argument to the *query* procedure). When invoked by all correct processes in set S sufficiently many times, with call *query*(S), I_{Ω^*} eventually permanently returns the same correct process in S (a leader) at all of these processes. More precisely, I_{Ω^*} ensures the following property in every execution e. Let S be a set of processes, such that every correct process in S invokes *query* infinitely many times in e, and $V \subseteq S$ be the set of all correct processes in S. Then, I_{Ω^*} guarantees that in execution e where (1) no process invokes *stop* infinitely many times and (2) all processes from set V eventually permanently pass set S as an argument to *query*: every process in V eventually returns the same process $p_l \in V$ in every call to *query*(S).

In the same vein, IFD $I_{\Diamond\mathcal{P}}$ is similar to $\Diamond\mathcal{P}$. $I_{\Diamond\mathcal{P}}$ ensures the following properties. Let V be a set of correct processes that, after some time, call *query* on $I_{\Diamond\mathcal{P}}$ and never call *stop* thereafter, and V' be a set of (correct) processes that call *query* and *stop* on $I_{\Diamond\mathcal{P}}$ infinitely many times. Call *query* invoked by a process p_i returns a set of processes *suspected* by p_i. $I_{\Diamond\mathcal{P}}$ guarantees that eventually: (1) every process in V suspects every crashed process, and (2) no process in V is ever suspected by any process in $V \cup V'$.

We establish a formal relationship between a failure detector $\mathcal{D}$ and its intermittent variant $I_{\mathcal{D}}$ by proving that the latter provides as much information about failures as the former. We do so by treating $I_{\mathcal{D}}$ as an abstract problem and proving that $\mathscr{D}$ is *the weakest failure detector* [6] to implement $I_{\mathscr{D}}$. We say then that $\mathscr{D}$ and $I_{\mathscr{D}}$ are *equivalent*. In the following two theorems we establish the relationship between I_{Ω^*} and Ω^*, and between $I_{\Diamond\mathscr{P}}$ and $\Diamond\mathscr{P}$.

Theorem 8 *I_{Ω^*} and Ω^* are equivalent.*

Proof We show that Ω^* is sufficient and necessary to implement I_{Ω^*}. The sufficiency part consists of exhibiting an algorithm that implements I_{Ω^*} using Ω^*. The necessity part consists of proving that the output of Ω^* can be emulated using some number of instances of I_{Ω^*} as "black boxes" and read-write registers.

It is easy to see that one can implement I_{Ω^*} using Ω^*. This can be done simply by making *query*(S), invoked by a process p_i, return the leader elected by Ω^* for set S, and ignoring every call to *stop*. Therefore, Ω^* is sufficient to implement I_{Ω^*}.

As Ω^* is a composition $\{\Omega_S\}_{S\subseteq\Pi,S\neq\emptyset}$, it is sufficient to prove for the necessity part that, for every non-empty subset S of set Π, there is an algorithm that extracts the output of Ω_S from any implementation of I_{Ω^*}.

Let L be any instance of I_{Ω^*} and let every alive process p_i periodically invoke *query*(S) on L and put the returned value in a local variable Ω_S*-output*$_i$. Also, assume no process ever invokes *stop* on L. Let V be the set of all correct processes. Clearly, every process in V will invoke *query*(S) infinitely many times. Furthermore, every correct process in S must belong to V. Thus, by the properties of I_{Ω^*}, every correct process in S has to eventually permanently output the id of the same correct process in S in variable Ω_S*-output*. Therefore, at every process p_i, Ω_S*-output*$_i$ is a valid output of failure detector Ω_S. Hence, for every $S \subseteq \Pi$, $S \neq \emptyset$, Ω_S is necessary to implement I_{Ω^*}, and so Ω^* is also necessary to implement I_{Ω^*}. □

Theorem 9 *$I_{\Diamond\mathscr{P}}$ and $\Diamond\mathscr{P}$ are equivalent.*

Proof We show that $\Diamond\mathscr{P}$ is sufficient and necessary to implement $I_{\Diamond\mathscr{P}}$. The sufficiency part consists of exhibiting an algorithm that implements $I_{\Diamond\mathscr{P}}$ using $\Diamond\mathscr{P}$. The necessity part consists of showing that the output of $\Diamond\mathscr{P}$ can be emulated using some number of instances of $I_{\Diamond\mathscr{P}}$ as "black boxes" and read-write registers.

It is easy to see that failure detector $\Diamond\mathscr{P}$ helps easily implement $I_{\Diamond\mathscr{P}}$. This can be done simply by making *query*, invoked by a process p_i, return the set of suspected processes output by $\Diamond\mathscr{P}$ at p_i, and ignoring every call to *stop*. Therefore, $\Diamond\mathscr{P}$ is sufficient to implement $I_{\Diamond\mathscr{P}}$.

Let D be any instance of $I_{\Diamond\mathscr{P}}$. Assume that every alive process p_i periodically invokes *query* on D and puts the returned value in a local variable $\Diamond\mathscr{P}$*-output*$_i$. Also, assume no process ever invokes *stop* on D. Let V be the set of all correct processes. Clearly, every process in V will invoke *query* infinitely many times and never invoke *stop*. Thus, by the properties of $I_{\Diamond\mathscr{P}}$, at every process p_i the variable $\Diamond\mathscr{P}$*-output*$_i$ is a valid output of failure detector $\Diamond\mathscr{P}$. Therefore, $\Diamond\mathscr{P}$ is necessary to implement $I_{\Diamond\mathscr{P}}$. □

6 Contention managers CM_{nb} and CM_{wf}

We present in this section a non-blocking contention manager CM_{nb} that uses IFD I_{Ω^*}, and a wait-free contention manager CM_{wf} that uses IFD $I_{\Diamond\mathscr{P}}$. Both contention managers stop their local IFD modules when no information about failures is needed. Moreover, when contention is low, both CM_{nb} and CM_{wf} can use any other contention manager *PCM* that satisfies the following property:

Termination. No process is blocked infinitely long.

Therefore, a contention manager *PCM* that provides good average case performance (in low-contention scenarios) can be combined with the worst-case guarantees of CM_{nb} (non-blockingness) or CM_{wf} (wait-freedom).

Algorithm 6: Implementation of non-blocking contention manager CM_{nb} (code for process p_i)

```
uses: $T[1,\ldots,n]$—array of single-bit registers,
      $I_{\Omega^*}$—intermittent failure detector, $PCM$—a contention
      manager that satisfies Termination (optional)
initially: $T[1,\ldots,n]$, $ts_i \leftarrow false$, $tries_i \leftarrow 0$

6.1  upon $try_i$ do
6.2      if $tries_i > maxTries$ then Serialize()
6.3      else
6.4          if $tries_i > 0$ then $PCM.try_i$
6.5          $tries_i \leftarrow tries_i + 1$

6.6  upon $resign_i$ do
6.7      if $ts_i$ then
6.8          $T[i] \leftarrow false$
6.9          $ts_i \leftarrow false$
6.10         $I_{\Omega^*}.stop$
6.11     $tries_i \leftarrow 0$

6.12 procedure Serialize()
6.13     if not $ts_i$ then
6.14         $ts_i \leftarrow true$
6.15         $T[i] \leftarrow true$
6.16     repeat
6.17         $S_i \leftarrow \{p_j \in \Pi \mid T[j] = true\}$
6.18     until $I_{\Omega^*}.query(S_i) = p_i$
```

The implementation of CM_{nb} is shown in Algorithm 6 and the underlying idea is the following. If a process p_i calls try_i more than *maxTries* times before resigning, this means that p_i cannot complete its current operation (*maxTries* is some natural constant). Thus, neither obstruction-freedom nor contention manager *PCM* is sufficient to provide progress for p_i anymore. In such case, p_i enters the serialization mechanism (procedure *Serialize*).

The role of the serialization mechanism is to prevent livelocks. Indeed, if after some time no active process is able to complete its operation, then all active processes will eventually enter the serialization mechanism (line 6.2) and only one of them, say process p_i, will be allowed to take steps (and run obstruction-free), while others will get blocked (lines 6.16–6.18). Once the chosen process (leader) p_i resigns, p_i announces this fact to blocked processes (in line 6.8) so that they can choose another active process among them as a leader. Also when p_i crashes, a new leader is elected.

The output of $\mathcal{I}_{\Omega^*}$ is used (in line 6.18) only by *serialized processes*, i.e., by every alive process p_j for which $T[j] = true$. This means that module $\mathcal{I}_{\Omega^*}$ can be suspended at each non-serialized process. That is why each serialized process p_j calls *stop* on $\mathcal{I}_{\Omega^*}$ when p_j resigns (line 6.10). Module $\mathcal{I}_{\Omega^*}$ starts working again on a process p_j once p_j invokes $query(S_j)$ again (in line 6.18).

The serialization mechanism lets only one active process take steps of an OF algorithm while blocking all others, but only when active processes eventually manage to chose a single leader among themselves in lines 6.16–6.18. If there is no agreement and so there are many leaders, none of them is guaranteed to be obstruction-free sufficiently long. If the elected leader crashes and active processes do not chose another leader, then it may happen that all active processes get blocked forever. Thus, the quality of the leader election provided by $\mathcal{I}_{\Omega^*}$ is vital and we need to explain why the limited properties guaranteed by $\mathcal{I}_{\Omega^*}$ are sufficient.

Intuitively, CM_{nb} guarantees non-blockingness when the leader election provided by $\mathcal{I}_{\Omega^*}$ is eventually accurate. However, $\mathcal{I}_{\Omega^*}$, as used by CM_{nb}, guarantees the accuracy of the leader election only in executions in which non-blockingness is violated. Thus, if there existed an execution of an OF algorithm combined with CM_{nb} in which non-blockingness did not hold, $\mathcal{I}_{\Omega^*}$ would have to guarantee eventually accurate leader election in this execution, in which case CM_{nb} would have to guarantee non-blockingness. Hence, such an execution is effectively impossible.

More precisely, if in an execution e non-blockingness is violated, this means that at some point in time t there are some correct active processes (a set V) and no process resigns thereafter. But then all these processes will keep querying $\mathcal{I}_{\Omega^*}$ forever, eventually permanently about the same set of processes S. Furthermore, no process ever stops $\mathcal{I}_{\Omega^*}$ after time t, for $\mathcal{I}_{\Omega^*}$ may be stopped only by a process that resigns. Thus, eventually $\mathcal{I}_{\Omega^*}$ will make processes in set V output a single correct active process as their leader from some point in time forever. The elected leader will then be eventually obstruction-free, in which case the leader has to eventually complete the operation it executes and resign—contradicting our assumption that no process resigns after time t.

Lemma 3 *Contention manager CM_{nb} shown in Algorithm 6 guarantees non-blockingness for every OF algorithm.*

Proof By contradiction, assume that in some execution e of an OF algorithm $\mathcal{A}$ combined with contention manager CM_{nb} non-blockingness is violated. This means that there exists time t, such that some correct processes are active at t and no process resigns after t. Let us denote by t' a point in time after t, such that (1) only correct processes are alive after t' and (2) no process that is idle at t' becomes active after t'. Let us denote by V the set of processes that are active at t'. Clearly, each process in V is permanently active from time t' forever.

For each correct process $p_i \notin V$ the value $T[i]$ is eventually permanently set to *false* after t', for p_i had to resign before t or p_i is never active in e, and p_i can set $T[i]$ to *true* (in line 6.15) only when p_i is active. Each faulty process must have crashed by t'. Therefore, there exists a time after which, for each process $p_i \notin V$, the value of $T[i]$ will not change.

Each process p_i in set V has to periodically invoke try_i, until p_i gets blocked forever (for execution e has to be well-formed). However, p_i can get blocked forever only in procedure *Serialize*, for *PCM* satisfies Termination. Thus, after t, each process in V will eventually enter procedure *Serialize* in line 6.2, because after t the value of $tries_i$ cannot be reset to 0 in line 6.11, as no process resigns after t, and $tries_i$ increases in line 6.5 each time p_i calls try_i and does not enter procedure *Serialize*. Thus, if at time t the value of $T[i]$ is *false*, p_i will eventually set the value to *true*. Furthermore, $T[i]$ cannot be reset to *false* after t as p_i does not resign after t. Therefore, there is a time $t'' > t'$, such that after t'': (1) for every process $p_i \in V$ the value $T[i]$ will be permanently set to *true*, and (2) for every process $p_j \notin V$ the value $T[j]$ will not change.

Denote by S the set of processes for which $T[\ldots] = true$ after time t''. Clearly, $V \subseteq S$ because for every process $p_i \in V$ the value of $T[i]$ is permanently set to *true* after t''. Also, no process invokes *stop* infinitely many times (in line 6.10) for no process resigns infinitely often in e. Furthermore, after time t'' only processes in set V will be querying $\mathcal{I}_{\Omega^*}$ in line 6.18, and eventually all processes in V will be querying $\mathcal{I}_{\Omega^*}$ about set S, constructed in line 6.17, which never changes after t''. Therefore, eventually, at each process p_i in set V, the module $\mathcal{I}_{\Omega^*}$ will be permanently returning the same process $p_l \in V$ in each call to $query(S_i = S)$ in line 6.18. Thus, eventually only process p_l will always return from procedure *Serialize*.

Therefore, eventually all processes in set V, except for p_l, will be blocked in lines 6.16–6.18 forever and process p_l will execute infinitely many steps of algorithm $\mathcal{A}$

obstruction-free. But then p_l, by obstruction-freedom of A, has to eventually resign and complete its current operation of A. Thus, $p_l \notin V$—a contradiction. □

From Theorem 1 and Lemma 3 we immediately obtain the following result:

Theorem 10 *Contention manager CM_{nb} is non-blocking.*

Algorithm 7: Implementation of wait-free contention manager CM_{wf} (code for process p_i)

```
uses: $S$—single-bit register, $T[1 \dots n]$—array of registers,
      $I_{\Diamond\mathcal{P}}$—intermittent failure detector, $PCM$—a contention
      manager that satisfies Termination (optional)
initially: $S \leftarrow false$, $T[1 \dots n]$, $ts_i \leftarrow \bot$, $tries_i \leftarrow 0$

7.1  upon $try_i$ do
7.2      if $tries_i > maxTries$ then $S \leftarrow true$
7.3      if $S$ then
7.4          $tries_i \leftarrow maxTries + 1$
7.5          Serialize()
7.6      else
7.7          if $tries_i > 0$ then $PCM.try_i$
7.8          $tries_i \leftarrow tries_i + 1$

7.9  upon $resign_i$ do
7.10     if $ts_i \neq \bot$ then
7.11         $T[i] \leftarrow \bot$
7.12         $ts_i \leftarrow \bot$
7.13         $S \leftarrow false$
7.14         $I_{\Diamond\mathcal{P}}.stop$
7.15     $tries_i \leftarrow 0$

7.16 procedure Serialize()
7.17     if $ts_i = \bot$ then
7.18         $ts_i \leftarrow GetTimestamp()$
7.19         $T[i] \leftarrow ts_i$
7.20     repeat
7.21         $sact_i \leftarrow \{j | T[j] \neq \bot \wedge j \notin I_{\Diamond\mathcal{P}}.query\}$
7.22         $leader_i \leftarrow \text{argmin}_{j \in sact_i} T[j]$
7.23     until $leader_i = i$
```

The implementation of CM_{wf} is presented in Algorithm 7. The algorithm relies on a (wait-free) function *GetTimestamp()* for generating unique timestamps such that if some process gets a timestamp *ts* then no process gets a timestamp lower than *ts* infinitely many times. Such a timestamping mechanism can be easily implemented with registers.

The basic idea of CM_{wf} is the following. When an active process p_i invokes try_i more than *maxTries* times, CM_{wf} sets flag S to *true* in line 7.2 and starts serializing all reported operations. As long as flag S is raised, every new process that invokes *try* enters immediately the *serialization mechanism* (procedure *Serialize*).

The serialization mechanism works as follows. First, p_i gets a timestamp in line 7.18 and announces the timestamp in array T in line 7.19. Then, using $I_{\Diamond\mathcal{P}}$, p_i periodically runs a leader election mechanism: the non-suspected process that announced the lowest timestamp in T is elected a leader. If p_i is a leader, p_i returns from the serialization mechanism.

$I_{\Diamond\mathcal{P}}$ guarantees that eventually the same correct active process is elected leader by all serialized processes (unless these processes resign before). The leader executes steps of the OF algorithm obstruction-free and so it eventually resigns. After doing so, the leader resets its timestamp in lines 7.11 and 7.12 so that the active process that currently has the lowest timestamp can become a leader thereafter. When a serialized process finishes its operation, it sets flag S to *false* in line 7.13. As a result, once all concurrent serialized operations are completed, the processes might fall back to some other, may be more pragmatic, contention management scheme (provided by contention manager *PCM*).

It might not be straightforward to see why the properties of $I_{\Diamond\mathcal{P}}$ are strong enough for the serialization mechanism. Similarly to I_{Ω^*}, IFD $I_{\Diamond\mathcal{P}}$, when used with contention manager CM_{wf}, provides useful information only in executions in which wait-freedom is violated. Consider then an execution e of an OF algorithm combined with CM_{wf}. If in e wait-freedom is violated, there are some correct processes (a set V) that are active from some point in time t forever. These processes will at some time query $I_{\Diamond\mathcal{P}}$ and never stop $I_{\Diamond\mathcal{P}}$ thereafter. But then, by properties of $I_{\Diamond\mathcal{P}}$, processes in set V will be eventually never suspected by any other active process. Thus, all the active processes have to eventually elect the correct process with the lowest timestamp (in V) as their leader and let the process run obstruction-free forever. But the leader will have to eventually complete its operation then, and so it will not be active forever—contradicting our assumption.

Lemma 4 *Contention manager CM_{wf} implemented by Algorithm 7 guarantees wait-freedom for all OF algorithms.*

Proof By contradiction, assume that in some execution e of some OF algorithm A combined with contention manager CM_{wf} there are some correct processes (a set V) that do not complete their operations, i.e., from some point in time they are active forever. By properties of OF algorithms, each process from set V has to invoke *try* infinitely many times, unless the process gets blocked forever. However, the latter can happen only after the process gets serialized (i.e., enters procedure *Serialize*) and after the process receives and announces its timestamp in line 7.18 and line 7.19, respectively. That is because contention manager *PCM* satisfies Termination and so *PCM* cannot block any process forever in line 7.7. □

Claim For every process p_j in set V there is a timestamp $ts_j^F \neq \bot$ such that eventually $ts_j = ts_j^F$ forever.

Proof Let us take some process p_j in set V and denote by t a point in time after which p_j is active forever. Clearly, after

t process p_j cannot reset its timestamp to $\perp$ (in line 7.12) because p_j does not resign after t. Thus, by the condition in line 7.17, once p_j receives a timestamp after time t, p_j will retain this timestamp forever.

Assume then, by contradiction, that p_j has its timestamp equal to $\perp$ from time t forever. But after t process p_j is permanently active and thus p_j eventually has to enter the serialization mechanism, after calling try_j at most $maxTries + 1$ times. But then p_j will receive a non-$\perp$ timestamp in line 7.18 after t—a contradiction. □

Let us denote by p_i the process having the lowest timestamp in $\{ts_k^F \mid p_k \in V\}$ (there is always one, and only one, such a process, by the claim proved before and because timestamps are unique). We will lead to a contradiction by showing that p_i has to eventually complete its current operation and resign.

Firstly, let us observe that all processes in set V will query $I_{\Diamond\mathcal{P}}$ in line 7.21 infinitely many times and after some time they will never invoke *stop* anymore in line 7.14. Therefore, by properties of $I_{\Diamond\mathcal{P}}$, eventually every process $p_j \in V$ will permanently suspect every crashed process and will never suspect any other process from set V anymore. Therefore, eventually all processes in set V, except for p_i, will be blocked forever in lines 7.20–7.23, because p_i is in set V and p_i has the lowest timestamp from all processes in set V.

Let us consider time t after which:

– the failure detection at processes in set V (provided by $I_{\Diamond\mathcal{P}}$) is already accurate,
– only correct processes are alive,
– p_i has already got its timestamp $ts_i = ts_i^F$ in line 7.18 and announced it in line 7.19, and
– all active processes other than p_i have timestamps larger than ts_i or equal to $\perp$.

The last condition will surely eventually hold in execution e because of the following reasons. First, timestamps are unique. Second, no process can get a timestamp lower than ts_i^F infinitely many times. Third, p_i has the lowest timestamp from all correct processes that never become idle after some point in time (set V) and so keep their once received timestamp forever.

Clearly, process p_i cannot be blocked infinitely long. Furthermore, all processes from set V, except for p_i, will eventually be blocked forever. This means that the only processes that can obstruct p_i infinitely many times (i.e., that can execute infinitely many steps of OF algorithm A concurrently with p_i) are these processes that complete infinitely many operations and thus call *try* and *resign* infinitely many times. Let us denote the set of these processes by V'. If we prove that V' is empty, then we show that from some point in time process p_i is obstruction-free forever and so, by obstruction-freedom, has to eventually complete its current operation and resign—a contradiction with our assumption that $p_i \in V$.

Claim Set V' is empty.

Proof Suppose not—that there are some processes that belong to V', i.e., processes that invoke *try* and *resign* infinitely many times. Process p_i sets flag S to *true* in line 7.2 infinitely many times, because p_i must execute line 7.4 after time t and cannot reset $tries_i$ thereafter. Therefore, there has to be some process $p_j \in V'$ that observes $S = true$ in line 7.3 and enters procedure *Serialize* in line 7.5 infinitely many times. This is because flag S can be reset to *false* only by a process that observes $S = true$ (and thus enters the serialization mechanism) and resigns, and S is set to *true* infinitely many times by p_i. Process p_j will then invoke *query* and *stop* on $I_{\Diamond\mathcal{P}}$ infinitely often. But p_j will always have a timestamp larger than ts_i^F after time t and, by properties of $I_{\Diamond\mathcal{P}}$, p_j will eventually never suspect process $p_i \in V$. Thus, eventually process p_j will be blocked forever and so $p_j \notin V'$—a contradiction. □

From Theorem 2 and Lemma 4 we immediately obtain the following result:

Theorem 11 *Contention manager CM_{wf} is wait-free.*

7 Implementation of IFDs I_{Ω^*} and $I_{\Diamond\mathcal{P}}$

Precisely because I_{Ω^*} and $I_{\Diamond\mathcal{P}}$ are sufficient to implement a non-blocking contention manager, they are impossible to implement in an asynchronous system. It is however usually reasonable to assume *eventual synchrony*, which means that eventually there exists an upper and a lower bound on the time it can take for a process to execute a step. These bounds are not known to processes, can be arbitrary and also can be different in each execution.

An example implementation of $I_{\Diamond\mathcal{P}}$ in an eventually synchronous system, similar to known message passing implementations of $\Diamond\mathcal{P}$ [1,7,9,25], is presented in Algorithm 8. The idea of the algorithm is the following. Each process p_i, for which IFD is not stopped, periodically increments a "heartbeat" register $A[i]$. Process p_i also checks the registers $A[\ldots]$ of other processes. If the value in a register $A[j]$ of process p_j has not changed since the last read, then p_i starts suspecting p_j (which means that a correct processes that never queries $I_{\Diamond\mathcal{P}}$ can be eventually permanently suspected). If p_i observes later that p_j has incremented its register, then p_i stops suspecting p_j and increases its *timeout* value. This timeout tells p_i how many steps p_i has to perform between two checks of the registers of other processes. Eventually p_i adjusts its timeout to the slowest process, provided that p_i is running $I_{\Diamond\mathcal{P}}$ for sufficiently long time.

Algorithm 8: Implementation of intermittent failure detector $I_{\Diamond\mathscr{P}}$ (code for process p_i)

```
uses: $A[1,\ldots,n]$—array of registers
initially: $A[1,\ldots,n] \leftarrow 1$, $prev_i[1,\ldots,n] \leftarrow 0$, $timeout_i \leftarrow$
      initial timeout, $output_i \leftarrow \emptyset$, $run_i \leftarrow false$
8.1  upon $run_i$ do
8.2      repeat
8.3          for $k \leftarrow 1$ to $timeout_i$ do $A[i] \leftarrow A[i]+1$
8.4          $suspected_i \leftarrow \emptyset$
8.5          for $j \leftarrow 1$ to $n$ do
8.6              if $prev_i[j] < A[j]$ then
8.7                  $prev_i[j] \leftarrow A[j]$
8.8                  if $j \in output_i$ then increase $timeout_i$
8.9              else $suspected_i \leftarrow suspected_i \cup \{p_j\}$
8.10         $output_i \leftarrow suspected_i$
8.11     until not $run_i$

8.12 upon query do
8.13     $run_i \leftarrow true$
8.14     return $output_i$

8.15 upon stop do
8.16     $run_i \leftarrow false$
```

Theorem 12 *Algorithm 8 implements $I_{\Diamond\mathscr{P}}$.*

Proof Let us denote by V the set of correct processes that at some point in time call *query* and never call *stop* thereafter. Let us denote by V' the set of processes that call *query* and *stop* infinitely many times. Let us take a point in time t, such that after t every process $p_j \in V$ has its value of run_j equal to *true* forever.

If a process p_i crashes, then p_i will no longer increment the value in $A[i]$ in line 8.3. As $run_j = true$ at every process $p_j \in V$ after t, all processes in V will eventually execute the "repeat" loop in lines 8.2–8.11 twice after the crash of p_i, and observe that $A[i]$ has not changed (in line 8.6). Thus, every process $p_j \in V$ will eventually add p_i to its set $suspected_j$ in line 8.9. Therefore, eventually p_i will be suspected by all processes in V and thus we have proved property 1 of $I_{\Diamond\mathscr{P}}$.

Now let us prove property 2. Assume, by contradiction, that a process $p_i \in V$ is suspected infinitely often by a process $p_j \in V \cup V'$. Process p_i is in V and so, after time t, $run_i = true$ forever. Therefore, p_i will increment its register $A[i]$ infinitely many times in line 8.3. Process p_j is in $V \cup V'$ and so the condition $run_j = true$ is satisfied infinitely many times, which means that p_j will execute the loop in lines 8.2–8.11 infinitely often. Therefore, p_j will observe in line 8.6 infinitely many times that $A[i]$ has changed and so, as p_j suspects p_i infinitely often, p_j will increase its timeout in line 8.8 infinitely many times. It means that at some point in time $timeout_j$ will be so large that p_j will spend much more time in the loop in line 8.3 (consisting of $timeout_j$ steps) than it will take p_i to execute the code in lines 8.4–8.10 and increment $A[i]$ at least once in line 8.3 ($2n+1$ steps, which is constant in any given execution). This is because eventually there exists an upper and a lower bound on the time it can take for any process to take a step, and thus also the relative speed of the processes p_i and p_j is eventually bounded. Therefore, between any two checks of p_j, p_i will manage to increment $A[i]$, and so p_i will not be ever suspected by p_j—a contradiction. □

IFD I_{Ω^*} can be implemented in a similar way. In fact, one can easily extract the output of I_{Ω^*} using $I_{\Diamond\mathscr{P}}$: $query(S)$ invoked on I_{Ω^*} would then return this alive (i.e., non-suspected by $I_{\Diamond\mathscr{P}}$) process in set S that has the lowest identifier. Clearly, I_{Ω^*} can be implemented in a more efficient way if $I_{\Diamond\mathscr{P}}$ is not used, for we can make only the elected leader send "heartbeat" signals to others, unlike in the presented implementation of $I_{\Diamond\mathscr{P}}$ in which every alive process for which IFD is not stopped has to keep incrementing its "heartbeat" counter.

A more effective implementation of $I_{\Diamond\mathscr{P}}$, assuming an eventually synchronous system, is presented in Algorithm 9. The idea is straightforward: an alive process p_i with the lowest identifier among the processes that participate in the leader election is elected (line 9.4). Then p_i permanently increments its register $A[i]$ to inform others that p_i is still alive (line 9.6). If a process p_j observes that $A[i]$ has not changed since the last read, p_j suspects p_i of having crashed and elects a new leader. If later p_j discovers that p_i is alive, p_j increases the $timeout_j$ value (line 9.8) which tells p_j how long p_j should wait (in line 9.9) between any two checks of a register $A[i]$.

Theorem 13 *Algorithm 9 implements I_{Ω^*}.*

Proof Assume, by contradiction, that Algorithm 9 does not implement I_{Ω^*}. This means that there exists some execution e in which the property of I_{Ω^*} is violated.

Denote by V the set of processes that invoke *query* infinitely many times in e. Assume that there exists a set S of processes such that: (1) all correct processes in S belong to V, and (2) starting from some time t, all processes in set V periodically invoke $query(S)$. Assume also that no process invokes *stop* infinitely many times in e. Let us denote by p_l the process from set V that has the lowest identifier. We will lead to a contradiction by showing that all processes in V have to eventually permanently return p_l in every call to $query(S)$.

All correct processes from set S are in V, and p_l has the lowest identifier from all processes in V. Therefore, every process from set S that has the identifier lower than the identifier of p_l eventually crashes in e. Therefore, eventually no process $p_i \in S$ such that $i < l$ (i and l are the identifiers of process p_i and p_l, respectively) will increment its register

Algorithm 9: Implementation of intermittent failure detector I_{Ω^*} (code for process p_i)

```
uses: $A[1,\ldots,n]$—array of registers
initially: $ld_i \leftarrow p_i$, $timeout_i \leftarrow$ initial timeout,
           $A[1,\ldots,n] \leftarrow 1$, $last_i[1,\ldots,n] \leftarrow 0$, $pset_i \leftarrow \emptyset$,
           $run_i \leftarrow false$
9.1  upon $run_i$ do
9.2      while $run_i$ do
9.3          $prevld_i \leftarrow ld_i$
9.4          $ld_i \leftarrow$ process $p_j \in pset_i$ with the lowest id $j$ such that
             $A[j] > last_i[j]$ and $j < i$, or $p_i$ if no such $p_j$ exists
9.5          $last_i[j] \leftarrow A[j]$
9.6          if $ld_i = p_i$ then $A[i] \leftarrow A[i]+1$
9.7          else
9.8              if $prevld_i \neq ld_i$ then increase $timeout_i$
9.9              wait for $timeout_i$ steps
9.10 upon $query(S)$ do
9.11     $run_i \leftarrow true$
9.12     if $S = pset_i$ then return $ld_i$
9.13     else
9.14         $pset_i \leftarrow S$
9.15         return $p_i$
9.16 upon $stop$ do
9.17     $run_i \leftarrow false$
```

$A[i]$. This means that process p_l will eventually permanently elect itself a leader in line 9.4, because p_l has the lowest id from all correct processes in $pset_l$ and eventually $pset_l$ is permanently equal to S (as $p_l \in V$). Therefore, eventually process p_l, after some time t', will be periodically incrementing its register $A[l]$ in line 9.6 and never wait in line 9.9 anymore.

Suppose some process $p_i \in V$, $i \neq l$, never permanently elects p_l as its leader. As p_l is permanently increasing its register $A[l]$ after time t', process p_i has to observe infinitely many times in line 9.4 that $A[l]$ has changed, elect p_l a leader and wait for $timeout_i$ steps in line 9.9. Process p_i also infinitely many times elects other process as its leader, and so p_i has to increment the value of $timeout_i$ infinitely many times in line 9.8.

Process p_l, after time t', executes a constant (for a given number of processes) number of steps between two increments of $A[l]$ in line 9.6. As the system is eventually synchronous, there is an upper bound $t_{\max}$ on the time between any two increments of $A[l]$ by process p_l. There is also (eventually) a lower bound, $t_{\min}$, on the time in which process p_i can execute a single step in line 9.9. Therefore, there exists such a value of $timeout_i$ that $t_{\min} \cdot timeout_i > t_{\max}$. Thus, eventually process p_i will have to wait in line 9.9 longer than it may take for p_l to increment $A[l]$. This means that eventually p_i will observe that $A[l] > last_i[l]$ in every execution of line 9.4 and so p_i will eventually permanently elect p_l as a leader—a contradiction. □

Algorithms 8 and 9 use an array A of n unbounded registers for simplicity. In fact, A can be replaced by an array of $2n^2$ single-bit registers $send_{ij}$, $i, j = 1,\ldots,n$. Instead of incrementing $A[i]$, a process p_i would set $send_{ij}$ to $true$ for all $j = 1,\ldots,n$, and instead of comparing $A[i]$ to $prev_j[i]$, process p_j would check whether $send_{ij}$ is $true$ and reset $send_{ij}$ to $false$. Such an optimized implementation of $I_{\Diamond\mathcal{P}}$ or I_{Ω^*} uses $O(n^2)$ memory.

8 Overhead of contention management

We discuss now the inherent overhead of contention management. In general, a process that executes operations implemented by an obstruction-free object implementation without any contention should call *try* only few times—ideally, once per operation. In fact, in executions without contention, obstruction-freedom is strong enough a liveness guarantee and a contention manager is not needed. Thus, it seems desirable to minimize the number of steps of a contention manager that a process executes within any operation, during which *try* has been invoked only once, at the beginning of the operation.

It is easy to see that when *try* is invoked for the first time within a given operation (if $maxTries > 0$), contention manager CM_{nb} makes no process access the intermittent failure detector or a shared object. On the contrary, a process calling *try* implemented by contention manager CM_{wf} always performs at least one step, accessing shared register S. Thus, CM_{nb} guarantees that no process will execute a step of the contention manager (in a call *try* or *resign*) in an operation, during which *try* is called only once. Contention manager CM_{wf}, however, will always make processes access shared objects (execute steps) inside every call *try*, even in executions with no contention. The following theorem states that this is inherent to any wait-free contention managers.

Theorem 14 *Let A be any OF algorithm. There is no wait-free contention manager CM that guarantees the following: For every execution e of A combined with CM, every process p_i, and every operation invoked at a time t and completed at a time t' in e by process p_i, if there is only one event try_i^{inv} in period (t, t') in $e|i$, then there is no step of CM in period (t, t') in $e|i$.*

Proof Assume, by contradiction, that such a contention manager *CM* exists. Consider two correct processes: p_1 and p_2, executing an OF algorithm A presented in Algorithm 1 (in Sect. 2.3). It is easy to verify that A is an OF algorithm.

To establish a contradiction, we construct an execution of A combined with *CM* in which correct process p_1 can never complete its operation, i.e., wait-freedom is violated. Let p_1 and p_2 be correct processes and consider the following execution e:

1. Process p_1 starts executing operation *of-getTimestamp* and reaches line 1.5.
2. Process p_1 executes, then, steps in lines 1.5–1.6 for some value of j and suspends its execution for some time.
3. Then process p_2 starts executing operation *of-getTimestamp*, completes the operation and resigns. Clearly, p_2, while executing the operation, is obstruction-free and thus eventually has to complete the operation.
4. Next, p_1 continues executing steps and observes in line 1.8 that $A[j] = 2$. Thus, p_1 is not able to complete the operation and has to start the next iteration of the "while" loop.
5. Steps 2–5 are repeated forever.

To see why steps 2–5 can be repeated infinitely many times in execution e, notice that process p_2 invokes try_2 only once per each operation p_2 executes in e. That is, p_2 invokes try_2 only when p_2 is idle. This means that, by our assumptions about CM, p_2 does not execute any step on behalf of CM in execution e. Therefore, $H_{\mathrm{OF}}(e)|2 = e|2$, and so for the module of contention manager CM executed at process p_2 execution e is indistinguishable from an execution $e' = e|2$ (in which only process p_2 is ever active). Thus, there is such an execution $e' = e|2$, in which process p_2 is never delayed by CM for sufficiently long time, so that p_1 could complete its operation and resign. Therefore, execution e of an OF algorithm combined with CM, in which wait-freedom is violated, exists—a contradiction with the assumption that CM is wait-free. □

9 Concluding remarks

It is often argued that contention is rare in many practical settings. Therefore, it is very appealing to design algorithms that are optimized for the case when processes do not obstruct each other. Ensuring progress in the worst-case scenarios is still important, though, but can be delegated to specialized contention management oracles that, once devised, may be reused for various object implementations. This approach reduces the programmer's problem of designing a wait-free (resp. non-blocking) algorithm to guaranteeing obstruction-freedom, which is commonly perceived as being easier.

In this paper, we determined the minimal failure information to implement contention managers that provide strong progress guarantees (non-blockingness or wait-freedom) for all obstruction-free algorithms. Namely, we proved that failure detectors Ω^* and $\Diamond\mathcal{P}$ are the weakest to implement any non-blocking and wait-free contention manager, respectively. The proofs include concrete contention manager implementations that use these failure detectors. These implementations are interesting in their own right.

We argued, however, that there is a drawback in building contention managers that use failure detectors. Namely, the very notion of a failure detector induces a systematic cost of detecting process crashes, even when no information about failures is needed, e.g., in executions with low contention, in which obstruction-freedom is strong enough to guarantee progress. Our solution to this problem is the abstraction of an intermittent failure detector, which is of independent interest. This abstraction encapsulates a failure detection mechanism that knows of, and responds to, contention manager demands, yet can still be described with axiomatic properties and compared precisely to a failure detector. We showed that two intermittent failure detectors described in this paper, which are equivalent to failure detectors Ω^* and $\Diamond\mathcal{P}$, can be indeed used in an implementation of a, respectively, non-blocking and wait-free contention manager.

The contention managers we present in this paper do not, by default, exhibit good average-case performance. In most cases, they would simply serialize all concurrent operations, including those that concern disjoint sets of memory locations. This does not exploit situations when such operations can be safely run in parallel. Fortunately, as we show through Algorithm 6 and 7, our contention managers can be easily composed with contention management strategies that perform well in the average case but do not ensure any worst-case guarantees (namely non-blockingness or wait-freedom) [27, 28].

Acknowledgments We are very grateful to Hagit Attiya, Maurice Herlihy, Bastian Pochon, Faith Fich, Victor Luchangco, Mark Moir and Nir Shavit for interesting discussions on the topic of this paper. We would also like to thank the anonymous reviewers for helpful comments.

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Proceedings of the International Symposium on Distributed Computing (DISC) (2001)
2. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with reads and writes in the absence of step contention. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC) (2005)
3. Attiya, H., Welch, J.L.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn. Wiley, New York (2004)
4. Bershad, B.N.: Practical considerations for non-blocking concurrent objects. In: Proceedings of the 14th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 264–273 (1993)
5. Boichat, R., Dutta, P., Frølund, S., Guerraoui, R.: Deconstructing Paxos. ACM SIGACT News Distributed Computing Column **34**(1), 47–67 (2003). Revised version of EPFL Technical Report 200106, January 2001
6. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)

7. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
8. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
9. Fetzer, C., Raynal, M., Tronel, F.: An adaptive failure detection protocol. In: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing (2001)
10. Fich, F., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free algorithms can be practically wait-free. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC) (2005)
11. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(3), 374–382 (1985)
12. Gafni, E., Lamport, L.: Disk Paxos. Distrib. Comput. **1**(16), 1–20 (2003)
13. Guerraoui, R.: Indulgent algorithms. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC) (2000)
14. Guerraoui, R., Herlihy, M., Kapałka, M., Pochon, B.: Robust contention management in software transactional memory. In: Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL); in conjunction with the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) (2005)
15. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC). LNCS, pp. 303–323. Springer, Heidelberg (2005)
16. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC) (2005)
17. Guerraoui, R., Schiper, A.: "Γ-accurate" failure detectors. In: Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG). Springer, Heidelberg (1996)
18. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
19. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 92–101 (2003)
20. Herlihy, M., Luchango, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 522–529 (2003)
21. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
22. Jayanti, P.: Robust wait-free hierarchies. J. ACM **44**(4), 592–614 (1997)
23. LaMarca, A.: A performance evaluation of lock-free synchronization protocols. In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 130–140 (1994)
24. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
25. Larrea, M., Fernández, A., Arévalo, S.: On the implementation of unreliable failure detectors in partially synchronous systems. IEEE Trans. Comput. **53**(7), 815–828 (2004)
26. Moir, M., Anderson, J.H.: Wait-free algorithms for fast, long-lived renaming. Sci. Comput. Program. **25** (1995)
27. Scherer III, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: Proceedings of the Workshop on Concurrency and Synchronization in Java Programs; in conjunction with the 23th Annual ACM Symposium on Principles of Distributed Computing (PODC) (2004)
28. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC) (2005)