Boosting Obstruction-Freedom with Low Overhead*

Rachid Guerraoui[1,2] Michał Kapałka[2] Petr Kouznetsov[3]

[1] Computer Science and Artificial Intelligence Laboratory, MIT
[2] School of Computer and Communication Sciences, EPFL
[3] Max Planck Institute for Software Systems

14th July 2006

Abstract

A *contention manager* is a shared memory abstraction that boosts progress of *obstruction-free* algorithms. In this paper, we study the problem of the overhead of *non-blocking* and *wait-free* contention managers which use the minimal possible information about failures. More specifically, we show that ensuring non-blockingness can be achieved with no overhead and that this is not possible when wait-freedom is to be guaranteed. We prove that ensuring wait-freedom has an inherent overhead that, however, can be made arbitrarily small. This shows an interesting "efficiency gap" separating non-blocking and wait-free implementations.

To support our claims we present two modular contention manager implementations, both using the minimal possible information about failures: one that ensures non-blockingness with no overhead, and one that ensures wait-freedom and which overhead can be arbitrarily reduced. Interestingly, these contention managers can be used only as a last resort, and can themselves be combined with more pragmatic ones that provide good average case performance.

At the heart of our cost-effective contention managers lies the notion of an *intermittent failure detector*, which we believe is interesting in its own right. Strictly speaking, this is not a failure detector in the classical sense, but rather an oracle that provides processes with information about failures (only) in *certain* executions. In particular, in executions with no contention, and when failure detection is not needed, its overhead is not incurred. We consider two such abstractions, which we show can be implemented with little system synchrony, and describe how they help implement our cost-effective contention managers.

Keywords: shared memory, obstruction-free, non-blocking, wait-free, contention manager, failure detector.

1 Introduction

This paper studies shared memory implementations of atomic (also called *linearizable* [16]) objects. More specifically, we pursue an approach that separates two concerns of such implementations [18]. The idea is to devise object implementations that always ensure linearizability but give progress guarantees only when there is no contention and, independently, to devise generic modules that could be applied to any such implementations and boost their progress. The object implementations are said to be *obstruction-free* (or OF, for short) and the progress boosters are called *contention managers* [18].

Obstruction-freedom is a liveness condition that requires every process to complete its operation in the eventual absence of interference from other operations [18]. This liveness condition appears to make sense in practice, because the absence of contention is typically argued to be the most common situation. In the presence of contention however, no progress is ensured and, in particular, processes might livelock. That is precisely where a contention manager helps. When a process cannot complete its operation for a long time, a contention manager might delay others so that the process can run alone sufficiently long and finish its task.

Ideally, a contention manager should ensure *wait-freedom*, i.e., progress for all, or at least *non-blockingness*, i.e., progress for at least one process,

*EPFL Technical Report LPD-REPORT-2006-xxx. Submitted for publication.

even in situations with high contention. However, it should incur no *overhead* in situations with no contention, for this would question the overall benefits of the obstruction-free implementation.

Achieving these objectives is not trivial, because every time an OF algorithm interacts with a contention manager, some overhead, even if negligible, is incurred. An ideal contention manager would be then one which is completely invisible (stopped) at every process that manages to complete its operations on its own, and which starts acting only at these processes that need help. In other words, the interaction between an OF algorithm and such a contention manager could be entirely suspended for all processes that execute steps alone, without loss of the provided liveness guarantees.

We show in this paper that for an important class of OF algorithms we can ensure non-blockingness in such an ideal way. On the contrary, guaranteeing wait-freedom requires that some overhead of contention management is incurred, even for processes that run in isolation. This inherent limitation of wait-freedom manifests itself not only in situations with no contention, but also in cases when processes are not able to observe that others run concurrently. Such situations are very frequent for *disjoint-access-parallel* [20] OF algorithms implementing large data structures, even if many processes execute operations at the same time.

In fact, if only non-blockingness is to be ensured, a contention manager, helping a group of processes that compete for some set of objects, does not have to delay, or in any way deal with, processes that run concurrently but do not themselves need any help. This means that a high degree of parallelism can be achieved, which is an important performance factor.

As pointed out in [10, 13], and proved in [15], ensuring progress requires information about failures. In short, a process that fails in the middle of an execution should be detected in order not to prevent other processes from progressing, but should not be confused with a process that is just slow completing its operation. Detecting failures is often accomplished using some timeout mechanism which requires some synchrony assumptions about the system. The contention managers presented in [10, 13] implement such timeout-based failure detection directly, using weak and only eventual synchrony assumptions. The approach leads, however, to non-modular implementations which are difficult to adapt to systems with other synchrony assumptions or with indications from the operating system about processes that have crashed or are paged-out [4].

An alternative is to encapsulate timing assumptions within abstract *failure detector* oracles [7]. In [15], we described contention managers that ensure wait-freedom and non-blockingness using failure detectors Ω^* and $\Diamond\mathcal{P}$, which we proved are in a precise sense minimal. In short, failure detector Ω^* outputs, for all processes of every subset of processes, a correct leader in the subset. Failure detector $\Diamond\mathcal{P}$ ensures that every process that does not crash eventually accurately detects all failures [7]. This approach has an obvious advantage of modularity, besides making it possible to precisely capture *minimal* information about failures. When a system provides stronger synchrony guarantees, or indications about crashed or paged-out processes, only a failure detector implementation needs to be changed, not a contention manager itself.

However, the approach might introduce a significant overhead, even in situations with no contention. This is because failure detector implementations typically use time-outs, relying on *heartbeat* signals that need to be exchanged even in situations with no contention. Ideally, one would like to pay the price of these exchanges only in situations with high contention. However, by definition, a failure detector cannot depend on computations being performed, i.e., on the actual execution [7]. This means that a failure detector has to guarantee its properties in every execution, including those in which there is no need for information about failures, or in which some properties of the failure detector are not necessary. This is an obvious overkill in the contention management context.

This paper addresses this issue by introducing the concept of an *intermittent failure detector (IFD)* abstraction. This is not a failure detector in the sense of [7], but an oracle that provides information about failures in certain executions. We call the abstraction *intermittent* because it can be stopped and restarted at any time, independently at each process. It is a distributed abstraction and each process has a local module of it. When the module is stopped at a process p_i, then p_i does not perform any steps of the implementation of the module (exchange signals). This is similar in spirit to [11]. The properties of the IFD abstraction are ensured only for processes that, after some point in time, restart their IFD module and never stop their module thereafter. We also make the IFD module return only the information that a process explicitly queried for, similarly to [9].

To support our claims that overhead can be eliminated if only non-blockingess is to be guaranteed, and can be made arbitrarily small if wait-freedom has to be ensured, even when only the minimal information about failures can be used, we present two contention manager implementations. The former ensures non-blockingness using an intermittent variant of Ω^* (denoted by I_{Ω^*}) and the latter ensures wait-freedom using an intermittent variant of $\Diamond\mathcal{P}$ (denoted by $I_{\Diamond\mathcal{P}}$). The former does not incur any overhead when obstruction-freedom is sufficient to provide progress. The overhead of the latter can be made arbitrarily low, at the cost of decreased efficiency of contention management. We also show how, using little system synchrony, one can implement the IFDs used by our contention managers.

It is important to note that I_{Ω^*} (or $I_{\Diamond\mathcal{P}}$) never give more information about failures than Ω^* (or $\Diamond\mathcal{P}$, respectively). Furthermore, having an implementation of I_{Ω^*} (or $I_{\Diamond\mathcal{P}}$) allows for implementing Ω^* (or $\Diamond\mathcal{P}$, respectively). This gives some notion of *equivalence* between failure detectors and their intermittent variants, which we define in this paper.

The solution presented so far, consisting of two independent modules: an IFD and a contention manager, can be further improved so that it has low overhead and good efficiency also in scenarios with *low* contention, when some very simple and cost-effective techniques can be used to ensure progress. This can be done by adding one more module: a contention manager that is more "pragmatic" in a sense that, using various heuristics or application-specific assumptions, can provide good average-case performance [24, 25, 14]. More precisely, when some contention is detected, a pragmatic contention manager can be invoked and given a chance to provide progress. The more costly mechanisms, which are used by our contention managers to guarantee progress in the worst case, are then used only as a last recourse, after everything else failed to resolve contention. In a sense, good average case performance and worst case guarantees can be provided at the same time.

It is worth noting that the contention managers presented in this paper are mainly to illustrate our claims. They are similar to the contention management algorithms shown in [10] and [13], both of which ensure wait-freedom and use timeout-based failure detection mechanisms directly. In fact, many ideas used there, and also in our contention managers, are much older and come from DLS [8] and Paxos [22] algorithms, ported later to shared memory systems [12, 5].

2 Preliminaries

Processes and Failure Detectors. We consider a set of n processes $\Pi = \{p_1, \ldots, p_n\}$ in a shared memory system [16, 21]. 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* [7, 6] 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 $\mathcal{D}$ by accessing local variable $\mathcal{D}$-*output*$_i$—the output of the module of $\mathcal{D}$ at process p_i. Failure detectors can be partially ordered according to the amount of information about failures they provide. A failure detector $\mathcal{D}$ is *weaker than a failure detector* $\mathcal{D}'$, and we write $\mathcal{D} \preceq \mathcal{D}'$, if there exists an algorithm (called a *reduction* algorithm) that transforms $\mathcal{D}'$ into $\mathcal{D}$. If $\mathcal{D} \preceq \mathcal{D}'$ but $\mathcal{D}' \not\preceq \mathcal{D}$, we say that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$, and we write $\mathcal{D} \prec \mathcal{D}'$.

Failure detector $\Diamond\mathcal{P}$ [7] 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.

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. Failure detector Ω^* [15] 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.

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 a minimal 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* [19, 3] 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 (in case p_i executes a step on behalf of CM) queries a failure detector.

Obstruction-freedom [18, 17] stipulates that if a process that invokes an operation op on object O and from some point in time executes steps of A alone[1], then it 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* [16] ensures that every correct process that invokes an operation eventually returns from the operation.

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. 1). 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.[2]

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 $resign_i^{inv}$ an invocation of call try_i and $resign_i$, respectively (at process p_i), and by try_i^{ret} and $resign_i^{ret}$—the corresponding responses.

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 at which the event took place. Every execution e induces a *history* $H(e)$ that includes only application events (invocations and responses of high-level operations). The corresponding *CM-history* $H_{\mathrm{CM}}(e)$ is the subsequence of e containing only application events and cm-events of the execution, and the corresponding *OF-history* $H_{\mathrm{OF}}(e)$ is the subsequence of e containing only application events, cm-events, and steps of A. For a sequence s of events, $s|i$ denotes the 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 $H_{\mathrm{CM}}(e)|i$ that occurred before t is try_i^{inv} or $resign_i^{inv}$. 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 $H_{\mathrm{CM}}(e)|i$ 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 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

[1]I.e., without encountering *step contention* [2].

[2]An example OF algorithm that uses this model of interaction with a contention manager is presented in the proof of Theorem 8.

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

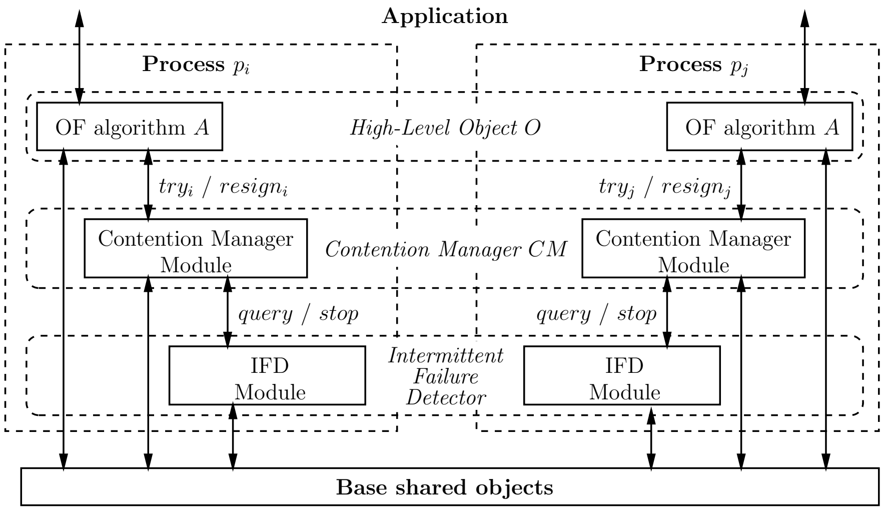

Figure 1: The OF algorithm/contention manager interface

after t or is obstruction-free in the interval $[t', \infty)$ for some $t' > t$. Note that, since algorithm A is obstruction-free, if an active process p_i is eventually obstruction-free, then p_i eventually resigns and completes its operation.

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 [19, 3], and, for every process p_i, (1) $H_{\mathrm{CM}}(e)|i$ 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^{ret}, …, try_i^{inv}, try_i^{ret}, $resign_i^{inv}$, $resign_i^{ret}$,$ret_i(op_k)$; (2) in $H_{\mathrm{OF}}(e)|i$, no step of A is executed when p_i is blocked or idle, (3) in $H_{\mathrm{OF}}(e)|i$, inv_i can only be followed by try_i^{inv}, and ret_i can only be preceded by $resign_i^{ret}$; (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*, ….

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.

A *non-blocking contention manager* guarantees non-blockingness for every OF algorithm. Intuitively, this will happen if the 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. More precisely, 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.

In [15] we prove the following theorem:

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

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 *wait-free contention* manager guarantees wait-freedom for every OF algorithm. Intuitively, this will happen if the contention manager makes sure that every correct active process is given "enough" time to complete its operation, regardless of how other processes behave. More precisely, 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.

In [15] we prove the following theorem:

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

Intermittent Failure Detectors. In this paper, we introduce two 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 calls that are used 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 *query*). 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 us denote by V the set of (correct) processes that invoke *query* infinitely many times and let $S \supseteq V$ be any set such that every correct process in S is also in V. I_{Ω^*} guarantees that if in execution e (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*, then every process in V will eventually return the same process $p_l \in V$ in every call to *query*(S).

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 which 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 $V \cup V'$.

[4]This property is ensured by wait-free contention managers from the literature [10, 13].

Comparing a Failure Detector with an Intermittent One. To establish a formal relationship between a failure detector $\mathcal{D}$ and its intermittent variant $I_{\mathcal{D}}$, we need to show that the latter provides as much information about failures as the former. We can do it by treating $I_{\mathcal{D}}$ as an abstract problem and proving that $\mathcal{D}$ is *the weakest failure detector* [6] to implement $I_{\mathcal{D}}$. If we prove this, we will say that $\mathcal{D}$ and $I_{\mathcal{D}}$ are *equivalent*. In the following two theorems we establish the relationship between I_{Ω^*} and Ω^* and between $I_{\Diamond\mathcal{P}}$ and $\Diamond\mathcal{P}$.

Theorem 3 *I_{Ω^*} and Ω^* are equivalent.*

Proof. To prove the theorem we need to show that Ω^* is sufficient and necessary to implement I_{Ω^*}. The sufficiency part consists of showing an algorithm that implements I_{Ω^*} using Ω^*. The necessity part has to show 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 having failure detector Ω^* one can easily implement I_{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 $\{\Omega_S\}_{S\subseteq\Pi, S\neq\emptyset}$, in the necessity part it is sufficient to prove that, for every non-empty subset S of set Π, there is an algorithm that extracts the output of Ω_S from an implementation of I_{Ω^*}.

Let L be an 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, let no process ever invoke *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^*}. $\square$

Theorem 4 *$I_{\Diamond\mathcal{P}}$ and $\Diamond\mathcal{P}$ are equivalent.*

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

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

Let D be an instance of $I_{\Diamond\mathcal{P}}$ and let every alive process p_i periodically invoke *query* on D and put the returned value in a local variable $\Diamond\mathcal{P}$*-output*$_i$. Also, let no process ever invoke *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 $I_{\Diamond\mathcal{P}}$, at every process p_i the variable $\Diamond\mathcal{P}$*-output*$_i$ is a valid output of failure detector $\Diamond\mathcal{P}$. Therefore, $\Diamond\mathcal{P}$ is necessary to implement $I_{\Diamond\mathcal{P}}$. $\square$

Overhead of Contention Management. We define the *overhead* of a contention manager CM in a non-empty execution e as the number of steps of CM in e divided by the number of operations invoked in e.

Let op_k be any operation executed by a process p_i, and t_{inv} and t_{ret} be the times of, respectively, the invocation and the return of op_k. Let us denote by try_i^0 the first call try_i invoked by p_i after time t_{inv}. If there is only one call try_i invoked by p_i in period $[t_{inv}, t_{ret}]$, then we will denote by $resign_i^0$ the call $resign_i$ invoked in $[t_{inv}, t_{ret}]$.

We say that CM guarantees a property Q (non-blockingness or wait-freedom) *with zero overhead* if CM guarantees Q in every execution of every OF algorithm combined with CM, even if every call try_i^0 and $resign_i^0$, for every process p_i, is substituted with an empty call[5].

Let A be any OF algorithm and CM—a contention manager. We say that an execution e of A combined with CM is *contention-free* if at every point in time there is at most one active process in e. We say that A is *contention-aware* if in every contention-free execution e of A combined with CM every call try_i, for every process p_i, is try_i^0. Thus, a process p_i executing a contention-aware algorithm calls try_i only at the beginning of an operation (try_i^0) and when p_i has not been obstruction-free since the invocation of the operation.

Therefore, if one has a contention manager CM that guarantees non-blockingness or wait-freedom with zero overhead, one can substitute all calls try_i^0 and $resign_i^0$ with empty calls (e.g., on compile time), thus making CM (as a whole, together with IFD and a "pragmatic" contention manager used by CM) invisible in all contention-free executions of contention-aware algorithms combined with CM.

It is important to note that most OF algorithms provide "for free" the ability to *eventually* detect that a process executing an operation has not been obstruction-free since try^0, and thus can be transformed into contention-aware ones in a straightforward way.

3 Non-Blocking Contention Manager

In this section we present a non-blocking contention manager CM_{nb} that uses IFD I_{Ω^*}. In short, the combined solution of CM_{nb} and I_{Ω^*} guarantees non-blockingness for every OF algorithm with zero overhead.

The implementation of CM_{nb} is shown in Algorithm 1 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 has problems completing 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 1.2) 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 1.16–1.18). Once the chosen process (leader) p_i resigns, p_i announces this fact to blocked processes (in line 1.8) so that they can choose another active process among them as a leader. Also when

[5]An empty call appears in an execution, but takes no time and is not visible to a contention manager.

p_i crashes, a new leader is elected.

The output of I_{Ω^*} is used (in line 1.18) only by *serialized processes*, i.e., by every alive process p_j for which $T[j] = true$. This means that module I_{Ω^*} can be suspended at each non-serialized process. That is why each serialized process p_j calls $stop$ on I_{Ω^*} when p_j resigns (line 1.10). Module I_{Ω^*} starts working again on a process p_j once p_j invokes $query(S_j)$ again (in line 1.18).

The serialization mechanism lets only one active process take steps of an OF algorithm while blocking all others only when active processes eventually manage to chose a single leader among themselves in lines 1.16–1.18. 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 process get blocked forever. Thus, the quality of the leader election provided by I_{Ω^*} is vital and we need to explain why the limited properties guaranteed by I_{Ω^*} are sufficient.

Intuitively, CM_{nb} guarantees non-blockingness when the leader election provided by I_{Ω^*} is eventually accurate. However, I_{Ω^*}, 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, I_{Ω^*} 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 I_{Ω^*} forever, eventually permanently about the same set of processes S. Furthermore, no process ever stops I_{Ω^*} after time t, for I_{Ω^*} may be stopped only by a process that resigns. Thus, eventually I_{Ω^*} 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 5 *Contention manager CM_{nb} shown in Algorithm 1 guarantees non-blockingness for every OF algorithm.*

Proof. By contradiction, let us assume that in some execution e of an OF algorithm 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 (a set V) and no process resigns after t. Let us denote by t' a point in time after t such that only correct processes are alive after t'.

For each correct process $p_i \notin V$ the value $T[i]$ is 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 1.15) only when p_i is active. Each faulty process must have crashed by $t' > t$. Therefore, for each process $p_i \notin V$, the value of $T[i]$ will not change after t'.

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 1.2 because after t the value of $tries_i$ cannot be reset to 0 in line 1.11, as no process resigns after t, and $tries_i$ increases in line 1.5 each time p_i calls try_i. 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, after some time $t'' > t' > t$, for every process $p_i \in V$, the value $T[i]$ will be permanently set to *true*. This means that after t'' no value of array T can change anymore.

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 1.10) for no process resigns infinitely often in e. Furthermore, after time t'' only processes in set V will be querying I_{Ω^*} in line 1.18 and eventually all processes in V will be querying I_{Ω^*} about set S, constructed in line 1.17, which never changes after t''. Therefore, eventually, at each process p_i in set V, the module I_{Ω^*} will be permanently returning the same process $p_l \in V$ in each call to $query(S_i = S)$ in line 1.18. 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 1.16–1.18 forever and process p_l will execute infinitely many steps of algorithm A obstruction-free. But then p_l, by obstruction-freedom of A, has to eventually complete its current operation of A and resign. Thus $p_l \notin V$—a contradiction. □

From Theorem 1 and Lemma 5 we immediately obtain a proof of the following theorem:

Theorem 6 *Contention manager CM_{nb} is non-blocking.*

It is easy to see that CM_{nb} guarantees non-blockingness with zero overhead. Indeed, the proof of Lemma 5 and Theorem 6 are still valid even under the assumption that every call try^0 and $resign^0$ is not visible to the contention manager. Thus, we immediately have the following result:

Theorem 7 *CM_{nb} guarantees non-blockingness with zero overhead.*

It is also worth noting that CM_{nb}, when used with contention-aware OF algorithms, allows for high degree of parallelism, without creating unnecessary hot spots. More precisely, if some process p_i, executing an operation *op* in period of time $[t, t']$, accesses some set of base shared objects B in steps of an OF algorithm, and no other process accesses any base shared object in set B in period $[t, t']$, then p_i will not get serialized and thus p_i will not execute any step of CM_{nb}. In particular, p_i will not access any base shared object, in *any* step in period $[t, t']$, which is accessed by other process in this period. Note that p_i does not have to be obstruction-free in $[t, t']$ for this to hold.

CM_{nb} uses only an array T, of size n, of single-bit registers. Therefore, it requires only bounded memory, unlike wait-free contention managers presented in [10, 13] and in the next section, which use unbounded timestamps. (We do not claim, though, that these contention managers could not use bounded timestamps—they just currently do not do it.)

When contention is low, CM_{nb} may use a pragmatic contention manager (satisfying Termination), denoted by *PCM*, to provide progress to processes. The border between low contention and high contention is determined by the *maxTries* constant: a non-blocking contention manager uses *PCM* at most *maxTries* times for any operation of an OF algorithm and later, if contention persists, starts to use serialization to guarantee non-blockingness.

4 Wait-Free Contention Manager

In this section we show that ensuring wait-freedom has an inherent overhead that cannot be completely eliminated. However, we present a wait-free contention manager, denoted by CM_{wf}, which uses the minimal possible information about failures and which overhead can be made arbitrarily small in contention-free executions.

Algorithm 3: An example OF algorithm implementing a timestamping mechanism

```
uses: $A[1,\ldots]$—unbounded array of registers,
      $B[1,\ldots]$—unbounded array of shared
      bits, $L$, $P$—registers
initially: $A[1,\ldots] \leftarrow \perp$, $B[1,\ldots] \leftarrow false$,
      $L \leftarrow 1$, $P \leftarrow \perp$
3.1  upon of-getTimestamp do
3.2      $CM.try_i$
3.3      $P \leftarrow i$
3.4      $j \leftarrow L$
3.5      while true do
3.6          $A[j] \leftarrow i$
3.7          if $B[j] = false$ then
3.8              $B[j] \leftarrow true$
3.9              if $A[j] = i$ then
3.10                 $L \leftarrow j$
3.11                 $CM.resign_i$
3.12                 return $j$
3.13             else $CM.try_i$
3.14         else if $P \neq i$ then
3.15             $CM.try_i$
3.16             $P \leftarrow i$
3.17         $j \leftarrow j + 1$
```

Theorem 8 *There is no contention manager that guarantees wait-freedom with zero overhead.*

Proof. Assume, by contradiction, that such a contention manager *CM* exists. This means that for every contention-free execution e of every contention-aware OF algorithm A combined with *CM*, wait-freedom has to be satisfied in e and no process can execute any step of *CM* (or IFD) in e.

Let us have two correct processes: p_i and p_j, executing an OF algorithm A presented in Algorithm 3. Algorithm A implements a timestamping mechanism and is based on the implementation of a splitter. It is easy to verify that A is a contention-aware OF algorithm.

Now let us take the following execution e of OF algorithm A combined with *CM* in which only processes p_i and p_k that are correct take steps of A:

1. Process p_i starts executing operation *of-getTimestamp* and reaches line 3.6.

Algorithm 1: Implementation of non-blocking contention manager CM_{nb}

```
uses: $T[1,\ldots,n]$—array of single-bit registers
initially: $T[1,\ldots,n], ts_i \leftarrow false, tries_i \leftarrow 0$

1.1  upon $try_i$ do
1.2      if $tries_i > maxTries$ then Serialize()
1.3      else
1.4          if $tries_i > 0$ then $PCM.try_i$
1.5          $tries_i \leftarrow tries_i + 1$

1.6  upon $resign_i$ do
1.7      if $ts_i$ then
1.8          $T[i] \leftarrow false$
1.9          $ts_i \leftarrow false$
1.10         $I_{\Omega^*}.stop$
1.11     $tries_i \leftarrow 0$

1.12 procedure Serialize()
1.13     if not $ts_i$ then
1.14         $ts_i \leftarrow true$
1.15         $T[i] \leftarrow true$
1.16     repeat
1.17         $S_i \leftarrow \{\, p_j \in \Pi \mid T[j] = true \,\}$
1.18     until $I_{\Omega^*}.query(S_i) = p_i$
```

I_{Ω^*}, $I_{\Diamond\mathcal{P}}$—intermittent failure detectors,
PCM—a contention manager that satisfies Termination (optional)

Algorithm 2: Implementation of wait-free contention manager CM_{wf}

```
uses: $S$—single-bit register, $T[1\ldots N]$—array
    of registers
initially: $S, T[1\ldots N], ts_i \leftarrow \perp, tries_i \leftarrow 0$

2.1  upon $try_i$ do
2.2      if $tries_i > maxTries$ then $S \leftarrow true$
2.3      if $S$ then
2.4          $tries_i \leftarrow maxTries + 1$
2.5          Serialize()
2.6      else
2.7          if $tries_i > 0$ then $PCM.try_i$
2.8          $tries_i \leftarrow tries_i + 1$

2.9  upon $resign_i$ do
2.10     if $ts_i \neq \perp$ then
2.11         $T[i] \leftarrow \perp$
2.12         $ts_i \leftarrow \perp$
2.13         $S \leftarrow false$
2.14         $I_{\Diamond\mathcal{P}}.stop$
2.15     $tries_i \leftarrow 0$

2.16 procedure Serialize()
2.17     if $ts_i = \perp$ then
2.18         $ts_i \leftarrow GetTimestamp()$
2.19         $T[i] \leftarrow ts_i$
2.20     repeat
2.21         $sact_i \leftarrow \{j \mid T[j] \neq \perp \wedge j \notin I_{\Diamond\mathcal{P}}.query\}$
2.22         $leader_i \leftarrow \text{argmin}_{j \in sact_i} T[j]$
2.23     until $leader_i = i$
```

2. Process p_i executes, then, steps in lines 3.6–3.7 for some j and suspends its execution for some time.

3. Then process p_k starts executing operation *of-getTimestamp*, completes the operation and resigns. Clearly, p_k, while executing the operation, is obstruction-free and thus eventually has to complete the operation.

4. Next, p_i continues executing steps and observes in line 3.9 that $A[j] = k \neq i$. Thus, p_i is not able to complete the operation and has to start the next iteration of the "while" loop.

5. Steps 2–5 are repeated until p_i is eventually obstruction-free.

Clearly, execution e is not contention-free. Now, let us take a contention-free execution e' in which all processes except for p_k are permanently idle and which satisfies the following condition: $e'|k = e|k$.

No process can execute any step of *CM* in execution e', p_k in particular. Thus, for the contention manager module executed by process p_k execution e' is indistinguishable from execution e. Thus, neither in e, nor in e' the contention manager module at process p_k can make p_k execute any step or query an IFD.

However, in execution e process p_k has to eventually be blocked for sufficiently long time so that p_i is able to complete its operation. Otherwise, wait-freedom could be violated. But then the *CM* module at process p_k has to communicate with p_i in order to determine when p_i resigns and turns idle, for the time that is necessary for p_i to complete its operation is not known and *CM* cannot even assume that this time is bounded. That is because *CM* must not use any synchrony assumptions directly and *CM* must not ever query any IFD in e. Hence, we reach a contradiction. □

The implementation of CM_{wf} is presented in Algorithm 2. 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 2.2 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 2.18 and announces the timestamp in array T in line 2.19. 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 2.11 and 2.12 so that the active process which 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 2.13. 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}. In e, wait-freedom is violated, so 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, the 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. For completeness we prove the following theorem:

Lemma 9 *Contention manager CM_{wf} implemented by Algorithm 2 guarantees wait-freedom for all OF algorithms.*

Proof. By contradiction, let us 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 2.18 and line 2.19, respectively. That is because contention manager *PCM* satisfies Termination and so *PCM* cannot block any process forever in line 2.7.

Claim 10 *For every process p_j in set V there is a timestamp $ts_j^F \neq \perp$ 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 2.12) because p_j does not resign after t. Thus, by the condition in line 2.17, once p_j receives a timestamp after time t, p_j will not get any new timestamp thereafter.

If p_j has its timestamp different than $\perp$ at time t, then clearly p_j will retain this timestamp forever. Assume then that p_j has its timestamp equal to $\perp$ at time t. But after t process p_j is permanently active and thus p_j eventually has to enter the serialization mechanism and receive a timestamp in line 2.18. Thus, eventually p_j will have some constant timestamp different than $\perp$ forever, for p_j cannot reset its timestamp to $\perp$ after time t. □

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 Claim 10 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 2.21 infinitely many times and after some time they will never invoke *stop* anymore in line 2.14. 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 eventually blocked forever in lines 2.20–2.23, for 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 2.18 and announced it in line 2.19, 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. Firstly, timestamps are unique. Secondly, no process can get a timestamp lower than ts_i^F infinitely many times. Thirdly, 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 forever. 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 forever (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 11 *Set V' is empty.*

Proof. Suppose not—that there are some processes which belong to V', i.e., processes which invoke *try* and *resign* infinitely many times. Process p_i sets flag S to *true* in line 2.2 infinitely many times, for p_i must execute line 2.4 after time t and cannot reset $tries_i$ thereafter. Therefore, there has to be some process $p_j \in V'$ which observes that $S = true$ in line 2.3 and enters procedure *Serialize* in line 2.5 infinitely many times. 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 9 we immediately obtain a proof of the following theorem:

Theorem 12 *Contention manager CM_{wf} is wait-free.*

It is easy to see that the overhead of CM_{wf} in the set of all contention-free executions is one step (the read of flag S in line 2.3), provided that the OF algorithm is contention-aware. Indeed, in contention-free executions no process is ever serialized and *PCM* is never used. Interestingly, the overhead of CM_{wf} in contention-free executions can be made arbitrarily small for all contention-aware OF algorithms. Indeed, a modified version of CM_{wf} in which only every k-th invocation of try_i is executed and the rest return immediately, still guarantees wait-freedom, as proved by the following theorem. Clearly, in this case the overhead of CM_{wf} in contention-free executions can be made arbitrarily small by making the value of k sufficiently large.

Theorem 13 *For any OF algorithm A and any natural number k, contention manager CM_{wf} guarantees wait-freedom for A even if, for every process p_i, only every k-th invocation of try_i is executed and all others return immediately.*

Proof. The proof is straightforward. Let us take Algorithm 2 and add the following code before line 2.2 (m is initially equal to 1):

Clearly, after this change, at every process p_i, all invocations of *try*, except every k-th, return immediately without making p_i execute any step. It is easy to see that even after this change the proof of Theorem 12 holds and thus the so modified CM_{wf} remains wait-free. Intuitively, that is because the added code can neither block any process infinitely long nor prevent the process from eventually getting serialized, and every process

Algorithm 4: Changes to CM_{wf} to ignore all but every k-th invocation of try_i

```
2.1 upon $try_i$ do
        if $m < k$ then
            $m \leftarrow m + 1$
            return
        else $m \leftarrow 1$
2.2     ...
```

Algorithm 5: Implementation of intermittent failure detector $I_{\Diamond\mathcal{P}}$

```
     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,
           $output_i \leftarrow \emptyset$, $run_i \leftarrow false$
5.1  upon $run_i$ do
5.2      repeat
5.3          for $k \leftarrow 1$ to $timeout_i$ do $A[i] \leftarrow A[i] + 1$
5.4          $suspected_i \leftarrow \emptyset$
5.5          for $j \leftarrow 1$ to $N$ do
5.6              if $prev_i[j] < A[j]$ then
5.7                  $prev_i[j] \leftarrow A[j]$
5.8                  if $j \in output_i$ then increase
                     $timeout_i$
5.9              else $suspected_i \leftarrow suspected_i \cup \{p_j\}$
5.10         $output_i \leftarrow suspected_i$
5.11     until not $run_i$

5.12 upon $query$ do
5.13     $run_i \leftarrow true$
5.14     return $output_i$

5.15 upon $stop$ do
5.16     $run_i \leftarrow false$
```

from set V or V' (see the proof) will keep invoking *try* until the process gets block forever inside the serialization mechanism. □

Similarly to CM_{nb}, contention manager CM_{wf} can use a pragmatic contention manager, denoted by *PCM*, in low-contention scenarios and switch to serialization only as a last recourse. Wait-freedom is guaranteed provided that *PCM* satisfies Termination.

5 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 or wait-free 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. As we assume that the global clock is discrete, no process can execute infinitely many steps in a finite time, and we can simply define eventual synchrony by stating that there exists an upper bound on the time it can take for a process to execute a step. This bound is 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 $\Diamond\mathcal{P}$ [7, 1, 9, 23], is presented in Algorithm 5. 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 which 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.

Theorem 14 *Algorithm 5 implements $I_{\Diamond\mathcal{P}}$.*

Proof. Let us denote by V the set of correct processes which 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 5.3. 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 5.2–5.11 twice and observe that $A[i]$ has not changed (in line 5.6) and so the processes will add p_i to their sets *suspected* in line 5.9. Therefore, eventually p_i will be suspected by all processes in V and thus we have proved property 1.

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 5.3. 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 5.2–5.11 infinitely often. Therefore, p_j will observe in line 5.6 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 5.8 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 5.3 (consisting of $timeout_j$ steps) than it will take p_i to execute the code in lines 5.4–5.10 and increment $A[i]$ at least once in line 5.3 ($2N + 1$ steps, which is constant in any given execution). This is because 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 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 $I_{\Diamond\mathcal{P}}$: $query(S)$ invoked on I_{Ω^*} would then return this alive (i.e., non-suspected by $I_{\Diamond\mathcal{P}}$) process in set S that has the lowest identifier. Clearly, I_{Ω^*} can be implemented in a more efficient way if $I_{\Diamond\mathcal{P}}$ is not used, for we can make only the elected leader send "heartbeat" signals to others, unlike in the presented implementation of $I_{\Diamond\mathcal{P}}$ in which every alive process for which IFD is not stopped has to keep incrementing its "heartbeat" counter.

A more optimal implementation of $I_{\Diamond\mathcal{P}}$, assuming an eventually synchronous system, is presented in Algorithm 6. The idea is straightforward: an alive process p_i with the lowest identifier among the processes that participate in leader election is elected a leader (line 6.4). Then p_i permanently increments its register $A[i]$ to inform others that p_i is still alive (line 6.6). 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 6.8) which tells p_j how long p_j should wait (in line 6.9) between any two checks of a register $A[i]$.

Theorem 15 *Algorithm 6 implements* I_{Ω^*}

Algorithm 6: Implementation of intermittent failure detector I_{Ω^*}

```
uses: $A[1,\dots,n]$—array of registers
initially: $ld_i \leftarrow p_i$, $timeout_i \leftarrow$ initial timeout,
        $A[1,\dots,n] \leftarrow 1$, $last_i[1,\dots,n] \leftarrow 0$,
        $pset_i \leftarrow \emptyset$, $run_i \leftarrow false$

6.1  upon $run_i$ do
6.2      while $run_i$ do
6.3          $prevld_i \leftarrow ld_i$
6.4          $ld_i \leftarrow$ 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
6.5          $last_i[j] \leftarrow A[j]$
6.6          if $ld_i = p_i$ then $A[i] \leftarrow A[i] + 1$
6.7          else
6.8              if $prevld_i \neq ld_i$ then increase
                 $timeout_i$
6.9              wait for $timeout_i$ steps

6.10 upon $query(S)$ do
6.11     $run_i \leftarrow true$
6.12     if $S = pset_i$ then return $ld_i$
6.13     else
6.14         $pset_i \leftarrow S$
6.15         return $p_i$

6.16 upon $stop$ do
6.17     $run_i \leftarrow false$
```

Proof. Let us denote Algorithm 6 by L and assume, by contradiction, that L does not implement I_{Ω^*}. This means that there exists some execution e in which the property of I_{Ω^*} is violated.

Let us denote by V the 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 $A[i]$. This means that process p_l will eventually permanently elect itself a leader in line 6.4, for p_l has the lowest timestamp 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 6.6 and never wait in line 6.9 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 6.4 that $A[l]$ has changed, elect p_l a leader and wait for $timeout_i$ steps in line 6.9. 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 6.8.

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 6.6. 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 a lower bound, t_{min}, on the time in which process p_i can execute a single step in line 6.9. 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 6.9 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 6.4 and so p_i will eventually permanently elect p_l as a leader—a contradiction. □

Algorithm 5 and Algorithm 6 use an array A of n unbounded registers for simplicity. In fact, A can be replaced by an array B of $2n^2$ single-bit registers. Instead of incrementing $A[i]$, process p_i would set $send_{ij}$ to *true* 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.

Acknowledgements

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.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the International Symposium on Distributed Computing (DISC'01)*, 2001.

[2] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, 2005.

[3] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Wiley, 2004.

[4] B. Bershad, D. Redell, and J. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the Fifth Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 223–233, October 1992.

[5] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *ACM SIGACT News Distributed Computing Column*, 34(1):47 – 67, March 2003. Revised version of EPFL Technical Report 200106, January 2001.

[6] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[8] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288 – 323, April 1988.

[9] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, 2001.

[10] F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, 2005.

[11] R. Friedman, A. Mostéfaoui, and M. Raynal. Asynchronous bounded lifetime failure detectors. *Information Processing Letters*, 2(94):85–91, April 2005.

[12] E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*, 1(16):1–20, 2003.

[13] R. Guerraoui, M. Herlihy, M. Kapałka, and B. Pochon. 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'05)*, October 2005.

[14] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[15] R. Guerraoui, M. Kapałka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. Technical report, EPFL, 2006.

[16] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[17] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

[18] M. Herlihy, V. Luchango, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, pages 522–529, 2003.

[19] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

[20] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

[21] P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.

[22] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[23] M. Larrea, A. Fernández, and S. Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers*, 53(7):815–828, 2004.

[24] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[25] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.