How to Elect a Leader Faster than a Tournament

Dan Alistarh
Microsoft Research
dan.alistarh@microsoft.com

Rati Gelashvili
MIT
gelash@mit.edu

Adrian Vladu
MIT
avladu@mit.edu

ABSTRACT

The problem of electing a leader from among n contenders is one of the fundamental questions in distributed computing. In its simplest formulation, the task is as follows: given n processors, all participants must eventually return a *win* or *lose* indication, such that a single contender may *win*. Despite a considerable amount of work on leader election, the following question is still open: can we elect a leader in an asynchronous fault-prone system faster than just running a $\Theta(\log n)$-time tournament, against a strong adaptive adversary?

In this paper, we answer this question in the affirmative, improving on a decades-old upper bound. We introduce two new algorithmic ideas to reduce the time complexity of electing a leader to $O(\log^* n)$, using $O(n^2)$ point-to-point messages. A non-trivial application of our algorithm is a new upper bound for the *tight renaming* problem, assigning n items to the n participants in expected $O(\log^2 n)$ time and $O(n^2)$ messages. We complement our results with lower bound of $\Omega(n^2)$ messages for solving these two problems, closing the question of their message complexity.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Theory, Algorithms

Keywords

Leader election; randomized algorithms; message-passing; asynchronous; message complexity; lower bound.

1. INTRODUCTION

The problem of picking a leader from among a set of n contenders in a fault-prone system is among the most well-studied questions in distributed computing. In its simplest form, *leader election (test-and-set)* [1] is stated as follows. Given n participating processors, each of the contenders must eventually return either a *win* or *lose* indication, with the property that a single participant may *win*. Leader election is one of a set of canonical problems, or *tasks*, whose solvability and complexity are the focus of distributed computing theory, along with *consensus (agreement)* [24, 26], *mutual exclusion* [15], *renaming* [10], or *task allocation (do-all)* [23]. These problems are usually considered in *asynchronous* models, such as message-passing or shared-memory [25].

We focus on leader election in the *asynchronous message-passing* model, in which each of n processors is connected to every other processor via a point-to-point channel. Communication is asynchronous, i.e., messages can be arbitrarily delayed. Moreover, local computation of processors is also performed in asynchronous steps. The scheduling of computation steps and message deliveries in the system is controlled by a *strong (adaptive) adversary*, which can examine local state, including random coin flips, and crash $t < n/2$ of the participants at any point during the computation. The natural complexity metrics are *message complexity*, i.e., total number of messages sent by the protocol, and *time complexity*, i.e. the number of times a processor relies on the adversary to schedule a computation step or to deliver messages.

Many fundamental results in distributed computing are related to the complexity of canonical tasks in asynchronous models. For example, Fisher, Lynch, and Patterson [16] showed that it is impossible to solve consensus deterministically in an asynchronous system if one of the n participants may fail by crashing. This deterministic impossibility extends to leader election [21]. Since the publication of the FLP result, a tremendous amount of research effort has been invested into overcoming this impossibility for canonical tasks. Seminal work by Ben-Or [13] showed that relaxing the problem specification to allow probabilistic termination can circumvent FLP, and obtain efficient distributed algorithms.

Consequently, the past three decades have seen a continuous quest to improve the randomized upper and lower bounds for canonical tasks, and in fact tight (or almost tight) complexity bounds are now known, against a strong adversary, for consensus [11, 5], mutual exclusion [19, 20, 18], renaming [4], and task allocation [14, 7].

For leader election against a strong adversary, the situation is different. The fastest known solution is more than

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.
ACM 978-1-4503-3617-8 /15/07 ...$15.00
DOI: http://dx.doi.org/10.1145/2767386.2767420.

two decades old [1], and is a *tournament tree*: build a binary tree with n leaves, and pair up the participants, which start at the leaves, into two-processor "matches," whose winner is decided by two-processor randomized consensus; winners of such matches continue to compete, while losers drop out, until a single winner prevails. Time complexity is logarithmic, as the winner has to communicate at each tree level. No time lower bounds are known. Despite significant recent interest and progress on this problem in weaker adversarial models [6, 3, 17], the question of whether a tournament is optimal when electing a leader against a strong adversary is surprisingly still open.

Contribution: In this paper, we show that it is possible to break the logarithmic barrier in the classic asynchronous message-passing model, against an adaptive adversary. We present a new randomized algorithm which elects a leader in expected $O(\log^* n)$ time, sending $O(n^2)$ messages.

The algorithm is based on two new ideas, which we briefly describe below. The general structure is rather simple: computation occurs in *phases*, where each phase is designed to drop as many participants as possible, while ensuring that at least one processor survives.

Consider a simple implementation: each processor flips a biased coin at the beginning of the phase, to decide whether to give up (value 0) or continue (value 1), and communicates its choice to others. If at least one processor out of the n_r participants in phase r flips 1, all processors which flipped 0 can safely drop from contention. We could aim for $o(\log n)$ iterations by setting the probabilities to obtain less than a constant fraction of survivors in each phase. Unfortunately, a strong adversary easily breaks such a strategy: since it can see the flips, it can schedule all the processors that flipped 0 to complete the phase *before* any processor that flipped 1, forcing *everyone* to continue.

Techniques: Our first algorithmic idea is a way to *hide* the processor coin flips during the phase, handicapping the adaptive adversary. In each phase, each processor first takes a "poison pill" (moves to *commit* state), and broadcasts this to all other processors. The processor then flips a biased local coin to decide whether to drop out of contention (*low* priority) or to take an "antidote" (*high* priority), broadcasts its new state, and checks the states of other processors. Crucially, if it has flipped low priority, and sees *any other processor* either in *commit* state or in *high priority* state, the processor returns *lose*. Otherwise, it survives to the next phase.

The above mechanics guarantee at least one survivor (in the unlikely event where all processors flip *low* priority, they all survive), but can lead to few survivors in each phase. The insight is that, to ensure many survivors, the adversary must examine the processors' coin flips. But to do so, the adversary must first allow it to take the poison pill (state *commit*). Crucially, any low-priority processor observing this *commit* state automatically drops out. We prove that, because of this catch-22, the adversarial scheduler can do no more than to let processors execute each phase sequentially, one-by-one, hoping that the first processor flipping high priority, which eliminates all later low-priority participants, comes as late as possible in the sequence.

Now we can bias the flips such that a group of at most $O(\sqrt{n_r})$ processors survive because they flipped high priority, and $O(\sqrt{n_r})$ processors survive because they did not observe any high priority. This choice of bias seems hard to improve, as it yields the perfect balance between the sizes of the two groups of survivors.

Our second algorithmic idea breaks this roadblock. Consider two extreme scenarios for a phase: first when all participants communicate with each other, leading to similar views and second, when processors see fragmented views, observing just a subset of other processors. In the first case, each processor can safely set a low probability of surviving. This does not work in the second case since processor views have a lot of variance. We exploit this variance to break symmetry. Our technical argument combines these two strategies such that we obtain at most $O(\log^2 n_r)$ expected survivors in a phase, under *any* scheduling.

The final algorithm has additional useful properties. It is *adaptive*, meaning that, if $k \leq n$ processors participate, its complexity becomes $O(\log^* k)$. Moreover, since most participants drop in the first round of broadcast, the message complexity is $O(kn)$, which we shall prove is asymptotically optimal.

Renaming: Based on these properties we design a message-optimal algorithm for *strong renaming*, which assigns distinct items (or names) labeled from 1 to n to the n processors, using expected $O(n^2)$ messages and $O(\log^2 n)$ time. We employ a simple strategy: each processor repeatedly picks a random name that it sees as available, announces it, and competes for it via an instance of leader election. If the processor wins, it returns the name; otherwise, it tries again. The algorithm can be seen as a balls-into-bins game, in which n balls are the processors and bins are the names. We need to characterize two parameters: the maximum number of trials by a single processor, and the maximum contention on a single bin, as they are linked with message and time complexity. The critical difficulty is that, since rounds are not synchronised, the bin occupancy views perceived by the processors are effectively under adversarial control and out-of-date or incoherent views can lead to wasted trials and increased contention on the bins.

Our task is to prove that, in fact, this balls-into-bins process is robust to the correlations and skews in the trial probability distributions caused by asynchrony. Our approach is to carefully bound the evolution of processors' views and their trial distributions as more and more trials are performed. Roughly, for $j \geq 1$, we split the execution into time intervals, where at most $n/2^{j-1}$ names are available at the beginning of the interval j, and focus on bounding the number of wasted trials in each interval. The main technical difficulty that we overcome is that the views corresponding to these trials could be highly correlated, as the adversary may delay messages to increase the probability of a collision.

Lower bound: We match the message complexity of our algorithms with an $\Omega(n^2)$ lower bound on the expected message complexity of any leader election or renaming algorithm. The intuition behind the bound is that no processor should be able to decide without receiving a message, as the others might have already elected a winner; since the adversary can fail up to $n/2$ processors, it should be able to force each processor to either send or receive $n/2$ messages. However, this intuition is not entirely correct, as groups of processors could employ complex gossip-like message distribution strategies to guarantee that at least *some* processors receive *some* messages while keeping the total message count

$o(n^2)$. We thwart such strategies with a non-trivial indistinguishability argument, showing that in fact there must exist a group of $\Theta(n)$ processors, each of which either sends or receives a total of $\Theta(n)$ messages. A similar argument yields the $\Omega(n^2)$ lower bound for renaming, and in fact for any object with strongly non-commutative operations [12].

Related Work: We focus on previous work on the complexity of randomized leader election[1] and renaming, most of which considered the asynchronous shared-memory. However, one option is to emulate efficient shared-memory solutions via simulations between shared-memory and message-passing [9]. This preserves time complexity, but communication may be increased by at most a linear factor.

We classify previous solutions according to their adversarial model. Against a strong adversary, the fastest known leader election algorithm is the tournament tree of Afek et al. [1], whose contention-adaptive variant was given in [6]. For n participants, these algorithms require $\Theta(\log n)$ time, and $\Theta(n^2 \log n)$ messages using a careful simulation. *PoisonPill* is contention-adaptive, improves time complexity (more than) exponentially, and gives tight message complexity bounds.

For renaming, fastest known shared-memory algorithm [4] can be simulated with $O(\log n)$ time, and $\Theta(n^2 \log n)$ messages. (The latter bounds are obtained by simulating an AKS sorting network [2]; constructible solutions pay an extra logarithmic factor in both measures.) Our balls-into-bins approach is simpler and message-optimal, at the cost of an extra logarithmic factor in the time complexity. Reference [6] uses a simpler balls-into-bins approach for renaming, where each processor tries all the names, in random order, until acquiring some one. Despite the similarity, this algorithm has expected time complexity $\Omega(n)$, as a late processor may try out a linear number of spots before succeeding.

References [3, 17] considered the complexity of leader election against a weak (oblivious) adversary, which fixes the schedule in advance. The structure of splitting the computation into sifting rounds, eliminating more than a constant factor of the participants per round, was introduced in [3], where the authors give an algorithm with $O(\log \log n)$ time complexity. Giakkoupis and Woelfel [17] improved this to $O(\log^* k)$, where k is the number of participants. These algorithms yield the same bounds in asynchronous message-passing, but their complexity bounds only hold against a *weak* adversary, which fixes the schedule in advance.

The *consensus* problem can be stated as leader election if we ask processors to return the *identifier* of the winner, as opposed to a win/lose indication. As such, consensus solves leader election, but not vice-versa [21]. In fact, randomized consensus has $\Omega(n)$ time complexity [11]. Recent work [5] considered the message complexity of randomized consensus in the same model, achieving $O(n^2 \log^2 n)$ message complexity, and $O(n \log^3 n)$ time complexity, using completely different techniques.

2. DEFINITIONS AND NOTATION

System Model: We consider classic asynchronous message-passing model [9]. Here, n processors communicate with each other by sending *messages* through *channels*. There is one channel from each processor to every other processor; the channel from i to j is independent from the channel from j to i. Messages can be arbitrarily delayed by a channel, but do not get corrupted.

Computations are modeled as sequences of steps of the processors, which can be either *delivery steps*, representing the delivery of a new message, or *computation steps*. At each computation step, the processor receives all messages delivered to it since the last computation step, and, unless it is *faulty*, it can perform local computation and send new messages. A processor is *non-faulty*, if it is allowed to perform local computations and send messages infinitely often and if all messages it sends are eventually delivered. Notice that messages are also delivered to *faulty* processors, although their outgoing messages may be dropped.

We consider algorithms that tolerate up to $t \leq \lceil n/2 \rceil - 1$ processor failures. That is, when more than half of the processors are non-faulty, they all return an answer from the protocol with probability one. A standard assumption in this setting is that all non-faulty processors always take part in the computation by replying to the messages, irrespective of whether they participate in a certain algorithm or even after they return a value—otherwise, the $t \leq \lceil n/2 \rceil - 1$ condition may be violated.

The Communicate Primitive: Our algorithms use a procedure called communicate, defined in [9] as a building block for asynchronous communication. The call communicate(m) sends the message m to all n processors and waits for at least $\lfloor n/2 \rfloor + 1$ acknowledgments before proceeding with the protocol. The communicate procedure can be viewed as a best-effort broadcast mechanism; its key property is that any two communicate calls intersect in at least one recipient. In the following, a processor i will communicate messages of the form ($propagate$,v_i) or ($collect$,v). For the first message type, each recipient j updates its view of the variable v and acknowledges by sending back an ACK message. In the second case, the acknowledgement is a pair (ACK,v_j) containing j's view of the variable for the receiving process. In both cases, processor i waits for $> n/2$ ACK replies before proceeding with its protocol. In the case of *collect*, the communicate call returns an array of at least $\lfloor n/2 \rfloor + 1$ views that were received.

Adversary: We consider strong adversarial setting where the scheduling of processor steps, message deliveries and processor failures are controlled by an adaptive adversary. At any point, the adversary can examine the system state, including the outcomes of random coin flips, and adjusts the scheduling accordingly.

Complexity Measures: We consider two worst-case complexity measures against the adaptive adversary. *Message complexity* is the maximum expected number of messages sent by all processors during an execution. When defining *time complexity*, we need to take into account the fact that, in asynchronous message-passing, the adversary schedules both message delivery and local computation.

Definition (Time Complexity)**.** *Assume that the adversary fixes two arbitrarily large numbers t_1 and t_2 before an execution, and these numbers are unknown to the algorithm. Then, during the execution, the adversary delivers every message of a non-faulty processor within time t_1 and sched-*

[1]Some older references, e.g. [1], employ the name *test-and-set* exclusively for this task, and use leader election for the consensus (agreement) problem, while more recent ones [17] equate test-and-set and leader election.

ules a subsequent step of any non-faulty processor in time at most t_2.[2] *An algorithm has time complexity $O(T)$ if the maximum expected time before all non-faulty processors return that the adversary can achieve is $O(T(t_1 + t_2))$.*[3]

For instance, in our algorithms, all messages are triggered by the communicate primitive. A processor depends on the adversary to schedule a step in order to compute and call communicate, and then depends on the adversary to deliver these messages and acknowledgments. In the above definition, if all processors call communicate at most T times, then all non-faulty processors return in time at most $2T(t_1+t_2) = O(T(t_1+t_2))$: each communicated message reaches destination in time t_1, gets processed within time t_2, at which point the acknowledgment is sent back and delivered after t_1 time. So, after $2t_1 + t_2$ time responses from more than half processors are received, and in at most t_2 time the next step of the processor is scheduled when it again computes and communicates. This implies the following.

Claim 2.1. *For any algorithm, if the maximum expected number of* communicate *calls by any processor that the adversary can achieve is $O(T)$, then time complexity is also $O(T)$.*

Problem Statements: In *leader election* (*test-and-set*), each processor may return either *WIN* or *LOSE*. Every (correct) processor should return (*termination*), and only one processor may return *WIN* (*unique winner*). No processor may lose before the eventual winner starts its execution. The goal is to ensure that operations are *linearizable*, i.e., can be ordered such that (1) the first operation is *WIN* and every other return value is *LOSE*, and (2) the order of non-overlapping operations is respected. *Strong (tight) renaming* requires every (correct) processor to eventually return a *unique* name between 1 and n.

3. LEADER ELECTION ALGORITHM

Our leader election algorithm guarantees that if k processors participate, the maximum expected number of communicate calls by any processor that the *strong adaptive* adversary can achieve is $O(\log^* k)$, and the maximum expected total number of messages is $O(nk)$. We start by illustrating the main algorithmic idea.

3.1 The PoisonPill Technique

Consider the protocol specified in Figure 1 from the point of view of a participating processor. The procedure receives the id of the processor as an input, and returns a *SURVIVE*/*DIE* indication. All n processors react to received messages by replying with acknowledgments according to the communicate procedure. In the following, we call a *quorum* any set of more than $n/2$ processors.

[2]Note that the adversary can set t_1 or t_2 arbitrarily large, unknown to the algorithm, so the guarantees from the algorithm's perspective are still only that messages are *eventually* delivered and steps are *eventually* scheduled.

[3]Applied to asynchronous shared-memory, this yields an alternative definition of *step (time) complexity*, taking t_2 as an upper bound on the time for a thread to take a shared-memory step (and ignoring t_1). Counting all the delivery and non-trivial computation *steps* in message-passing gives an alternative definition of message complexity, corresponding to shared-memory *work complexity*.

Each processor stores the current status of other processors in a *Status* array, which we also call its *view*. During the round, the processor will ask all other processors for their *Status* array. The contents are merged into the *Views* matrix, where $Views[i][j]$ is processor j's status, as seen by processor i.

```
   Input: Unique identifier i of the participating processor
   Output: SURVIVE or DIE
   Local variables:
       Status[n] = {⊥};
       Views[n][n];
       int coin;
1  procedure PoisonPill⟨i⟩
2      Status[i] ← Commit
                                   /* commit to coin flip */
3      communicate(propagate, Status[i])
                                   /* propagate status */
4      coin ← random(1 with probability 1/√n, 0 otherwise)
                                   /* flip coin */
5      if coin = 0 then Status[i] ← Low-Pri
6      else Status[i] ← High-Pri
7      communicate(propagate, Status[i])
                                   /* propagate updated status */
8      Views ← communicate(collect, Status)
                                   /* collect status from > n/2 */
9      if Status[i] = Low-Pri then
10         if ∃proc. j : (∃k : Views[k][j] ∈ {Commit, High-Pri}
           and ∀k' : Views[k'][j] ≠ Low-Pri) then
11             return DIE
               /* i has low priority, sees processor j
                  with either high priority or
                  committed and not low priority, and
                  dies */
12     return SURVIVE
```

Figure 1: PoisonPill **Technique**

Each participating processor first announces that it is about to flip a random coin by moving to state *Commit* (lines 2-3), then obtains either low or high priority based on the outcome of a biased coin flip. (As the name suggests, a high priority processor will survive the round, whereas a low priority may drop from contention in the round.) The processor then propagates its priority information to a quorum via the communicate procedure (line 7). More precisely, the processor i sends messages to all other processors, asking them to update their $Status[i]$ entries to the new value, and awaits acknowledgments from a quorum.

Next, processor i collects the status of other processors from a quorum using the communicate$(collect, Status)$ call on line 8, which requests views of the *Status* array from each processor j. Note that the resulting *Views* matrix will contain replies (rows) from at least $n/2$ processors.

The crux of the round procedure is the *DIE* condition on line 11. A processor i returns *DIE* at this line if *both* of the following occur: (1) the processor i has low priority, *and* (2) it observes another processor j that does not have low priority in any of the views, but j has either high priority or is committed to flipping a coin (state *Commit*) in some view. Otherwise, processor i survives. The first key observation is that

Claim 3.1. *If all processors participating in* PoisonPill *return, at least one processor survives.*

Proof. Assume the contrary. Since processors with high priority always survive, all participating processors must have

a low priority. All participants propagate their low priority information to a quorum by calling the communicate procedure on line 7. Let i be the last processor that completes this communicate call. At this point, the status information of all participants is already propagated to a quorum. More precisely, for every participating processor j, more than half of the processors have a view $Status[j] = Low\text{-}Pri$.

Therefore, when processor i proceeds to line 8 and collects the *Status* arrays from more than half of the processors, then, since any two quorums intersect, for every participating processor j, there will be a view of some processor k' showing j's low priority. All non-participating processors will have priority $\perp$ in all views. But given the structure of the protocol, processor i will *not* return on line 11 and will survive. This contradiction completes the proof. □

On the other hand, we can bound the expected number of processors that survive:

Claim 3.2. *The expected number of processors that return SURVIVE is $O(\sqrt{n})$.*

Proof. Consider the random coin flips on line 4 and let us highlight the first time when some processor i flips value 1. We will argue that no other processor j that subsequently (or simultaneously) flips value 0 can survive. Consider such a state. When processor j flips 0, processor i has already propagated its *Commit* status to a quorum of processors. Furthermore, processor i has a high priority, thus no processor can ever view it as having a low priority. Hence, when processor j collects views from a quorum, because every two quorums have an intersection, some processor k will definitely report the status of processor i as *Commit* or *High-Pri* and no processor will report *Low-Pri*. Thus, processor j will have to return *DIE* on line 11.

The above argument implies that processors survive either if they flip 1 and get a high priority, or if they flip 0 strictly before any other processor flips 1. Each of the at most n processors independently flips a biased coin on line 4 and hence, the number of processors that flip 1 is at most the number of 1's in n Bernoulli trials with success probability $1/\sqrt{n}$, in expectation $\sqrt{n}$. Processors that flip 0 at the same time as the first 1 do not survive, and it also takes $\sqrt{n}$ trials in expectation before the first 1 is flipped, giving an upper bound $\sqrt{n}$ on the maximum expected number of processors that can flip 0 and survive. □

It is possible to apply this technique recursively with some extra care and construct an algorithm with an expected $O(\log \log n)$ time complexity. But we do not want to stop here.

3.2 Heterogeneous PoisonPill

Building a more efficient algorithm based on the PoisonPill technique requires reducing the number of survivors beyond $\Omega(\sqrt{n})$ without violating the invariant that not all participants may die. We control the coin flip bias, but setting the probability of flipping 1 to $1/\sqrt{n}$ is provably optimal. Let the adversary schedule processors to execute PoisonPill sequentially. With a larger probability of flipping 1, more than $\sqrt{n}$ processors are expected to get a high priority and survive. With a smaller probability, at least the first $\sqrt{n}$ processors are expected to all have low priority and survive. There are always $\Omega(\sqrt{n})$ survivors.

To overcome the above lower bound, after committing, we make each processor record the list ℓ of all processors including itself, that have a non-$\perp$ status in some view collected from the quorum. Then we use the size of list ℓ of a processor to determine its probability bias. Each processor also augments priority with its ℓ and propagates that as a status. This way, every time a high or low priority of a processor p is observed, ℓ of processor p is also known. Finally, the survival criterion is modified: each processor first computes set L as the union of all processors whose non-$\perp$ statuses it ever observed itself, and of the ℓ lists it has observed in priority informations in these statuses. If there is a processor in L for which no reported view has low priority, the current processor drops.

The algorithm is described in Figure 2. The particular choice of coin flip bias is influenced by factors that should become clear from the analysis. Despite modifications, the same argument as in Claim 3.1 still guarantees at least one survivor. Let us now prove that the views of the processors have the following interesting *closure property*, which will be critical to bounding the number of survivors with low priority.

```
   Input: Unique identifier i of the participating processor
   Output: SURVIVE or DIE
   Local variables:
       Status[n] = {⊥};
       Views[n][n];
       int coin;
1  procedure HeterogeneousPoisonPill⟨i⟩
2      Status[i] ← {.stat = Commit, .list = {}}
                                   /* commit to coin flip */
3      communicate(propagate, Status[i])
                                     /* propagate status */
4      Views ← communicate(collect, Status)
                                /* collect status from > n/2 */
5      ℓ ← {j | ∃k : Views[k][j] ≠ ⊥}
                                   /* record participants */
6      if |ℓ| = 1 then prob ← 1                /* set bias */
7      else prob ← log|ℓ| / |ℓ|                /* set bias */
8      coin ← random(1 with probability prob, 0 otherwise)
                                                /* flip coin */
9      if coin = 0 then  Status[i] ← {.stat = Low-Pri, .list = ℓ}
                          /* record priority and list */
10     else  Status[i] ← {.stat = High-Pri, .list = ℓ}
                          /* record priority and list */
11     communicate(propagate, Status[i])
                        /* propagate priority and list */
12     Views ← communicate(collect, Status)
                                /* collect status from > n/2 */
13     if Status[i].stat = Low-Pri then
14         L ← ∪_{k,j: Views[k][j]≠⊥} Views[k][j].list
                                /* union all observed lists */
15         L ← L ∪ {j | ∃k : Views[k][j] ≠ ⊥}
                                  /* record new participants */
16         if ∃proc. j ∈ L : ∀k : Views[k][j].stat ≠ Low-Pri
           then
17             return DIE
               /* i has low priority, learns about
                  processor j participating whose low
                  priority is not reported, and dies
                  */
18     return SURVIVE
```

Figure 2: Heterogeneous PoisonPill

Claim 3.3. *Consider any set S of processors that each flip 0 and survive. Let U be the union of all L lists of processors in S. Then, for $p \in U$ and every processor q in the ℓ list of p, q is also in U.*

Proof. In order for processors in S to survive, they should have observed a low priority for each of the processors in their L lists. Thus, every processor $p \in U$ must flip 0, as otherwise it would not have a low priority. However, the low priority of p observed by a survivor was augmented by the ℓ list of p. According to the algorithm, the survivor includes in its own L all processors q from this ℓ list of p, implying $q \in U$. □

Next, let us prove a few other useful claims:

Claim 3.4. *If processor q completed executing line 3 no later than processor p completed executing line 3, then q will be included in the ℓ list of p.*

Proof. When p collects statuses on line 4 from a quorum, q is already done propagating its *Commit* on line 3. As every two quorum has an intersection, p will observe a non-$\perp$ status of q on line 5. □

Claim 3.5. *The probability of at least z processors flipping 0 and surviving is $O(1/z)$.*

Proof. Let S be the set of the z processors that flip 0 and survive and let us define U as in Claim 3.3. For any processor $p \in U$ and any processor q that completes executing line 3 no later than p, by Claim 3.4 processor q has to be contained in the ℓ list of p, which by the closure property (Claim 3.3) implies $q \in U$. Thus, if we consider the ordering of processors according to the time they complete executing line 3, all processors not in U must be ordered strictly after all processors in U.

Therefore, during the execution, first $|U|$ processors that complete line 3 must all flip 0. The adversary may influence the composition of U, but by the closure property, each ℓ list of processors in U contains only processors in U, meaning $|\ell| \leq |U|$. So the probability for each processor to flip 0 is at most $(1 - \frac{\log |U|}{|U|})$ and for all processors in U to flip 0's is at most $(1 - \frac{\log |U|}{|U|})^{|U|} = O(1/|U|)$. This is $O(1/z)$ since all z survivors from S are included in their own lists and hence also in U. □

We have never relied on knowing n. If $k \leq n$ processors participate in the heterogeneous PoisonPill, we get

Lemma 3.6. *The maximum expected number of processors that flip 0 and survive is $O(\log k) + O(1)$.*

Lemma 3.7. *The maximum expected number of processors that flip 1 is $O(\log^2 k) + O(1)$.*

Proof. Consider the ordering of processors according to the time they complete executing line 3, breaking ties arbitrarily. Due to Claim 3.4, the processor that is ordered first always has $|l| \geq 1$, the second processor always computes $|l| \geq 2$, and so on. The probability of flipping 1 decreases as $|l|$ increases, and the best expectation achievable by adversary is $1 + \sum_{l=2}^{k} \frac{\log l}{l} = O(\log^2 k) + O(1)$ as desired. □

3.3 Final construction

The idea of implementing leader election is to have rounds of heterogeneous PoisonPill, where all processors participate in the first round and only the survivors of round r participate in round $r + 1$. Each processor p, before participating in round r_p, first propagates r_p as its current round number to a quorum, then collects information about the rounds of other processors from a quorum. Let R be the maximum round number of a processor in all views that p collected. To determine the winner, we use the idea from [27]: if $R > r_p$, then p loses and if $R < r_p - 1$ then p wins. We also use a standard doorway mechanism [1] to ensure linearizability. The pseudocode of the final construction is given in the full version of this paper [8], along with the complete proof of the following statement:

Theorem 3.8. *Our leader election algorithm is linearizable. If there are at most $\lceil n/2 \rceil - 1$ processor faults, all non-faulty processors terminate with probability 1. For k participants, it has time complexity $O(\log^* k)$ and message complexity $O(kn)$.*

The performance guarantees follow from Lemma 3.6 and from Lemma 3.7 with some careful analysis. In particular, later rounds in which maximum expected number of participants is constant require special treatment.

4. THE RENAMING ALGORITHM

```
   Input: Unique identifier i from a large namespace
   Output: int name ∈ [n]
   Local variables:
       bool Contended[n] = {false};
       bool Views[n][n];
       int coin, spot, outcome;
1  procedure getName⟨i⟩
2      while true do
3          Views ← communicate(collect, Contended)
           /* collect contention information */
4          for j ← 1 to n do
5              if ∃k : Views[k][j] = true then
6                  Contended[j] ← true
                   /* mark names that became contended
                      */
7          communicate(propagate, {Contended[j] |
           Contended[j] = true})            /* propagate */
8          spot ← random(j | Contended[j] = false)
                     /* pick random uncontended name */
9          Contended[spot] ← true
10         outcome ← LeaderElect_spot(i)
                           /* contend for a new name */
11         communicate(propagate, Contended[spot])
                            /* propagate contention */
12         if outcome = WIN then
13             return spot  /* win iff you are leader */
```

Figure 3: Pseudocode of the renaming algorithm for n processors.

The algorithm is described in Figure 3. There is a separate leader election protocol for each name; which a processor must win in order to claim the name. Each processor repeatedly chooses a new name and contends for it by participating in the corresponding leader election, until it eventually wins. Processors keep track of contended names and use this information to choose the next name to compete for: in

particular, the next name is selected uniformly at random from the uncontended names.

We now give a detailed overview of the proof. The full proof is given in the technical report version of our paper [8]. First, we show that the algorithm is correct.

Lemma 4.1. *No two processors return the same name from the* getName *call and if there are at most* $\lceil n/2 \rceil - 1$ *processor faults, all non-faulty processors terminate with probability 1.*

Let us now introduce some notation. For an arbitrary execution, and for every name u, consider the first time when more than half of processors have $Contended[u] = \mathsf{true}$ in their view (or time ∞, if this never happens). Let $\prec$ denote the name ordering based on these times, and let $\{u_i\}$ be the sequence of names sorted according to increasing $\prec$. Among the names with time ∞, sort later the ones that are never contended by the processors. Resolve all the remaining ties according to the order of the names. This ordering has the following useful temporal property.

Lemma 4.2. *In any execution, if a processor views* $Contended[i] = \mathsf{true}$ *in some while loop iteration, and in some subsequent iteration on line 8 the same processor views* $Contended[j] = \mathsf{false}$*,* $i \prec j$ *has to hold.*

Let X_i be a random variable, denoting the number of processors that ever contend in a leader election for the name u_i. The following holds.

Lemma 4.3. *The message complexity of our renaming algorithm is* $O(n \cdot \mathbb{E}[\sum_{i=1}^{n} X_i])$.

We partition names $\{u_i\}$ into $\log n$ *groups*, where the first group G_1 contains the first $n/2$ names, second group G_2 contains the next $n/4$ names, etc. We use notation $G_{j' \geq j}$, $G_{j'' > j}$ and $G_{j''' < j}$ to denote the union of all groups $G_{j'}$ where $j' \geq j$, all groups $G_{j''}$ where $j'' > j$, and all groups $G_{j'''}$ where $j''' < j$, respectively. We can now split any execution into at most $\log n$ phases. The first phase starts when the execution starts and ends as soon as for each $u_i \in G_1$ more than half of the processors view $Contended[u_i] = \mathsf{true}$ (the way $\{u\}$ is sorted, this is the same as when the contention information about $u_{n/2}$ is propagated to a quorum). At this time, the second phase starts and ends when for each $u_i \in G_2$ more than half of the processors view $Contended[u_i] = \mathsf{true}$. When the second phase ends, the third phase starts, and so on.

Consider any loop iteration of some processor p in some execution. We say that an iteration *starts* at a time instant when p executes line 2 and reaches line 3. Let V_p be p's view of the *Contended* array right before picking a spot on line 8 in the given iteration. We say that an iteration is *clean*(j), if the iteration starts during phase j and no name from later groups $G_{j'' > j}$ is contended in V_p. We say that an iteration is *dirty*(j), if the iteration starts during phase j and some name from a later group $G_{j'' > j}$ is contended in V_p.

Observe that any iteration that starts in phase j can be uniquely classified as *clean*(j) or *dirty*(j) and in these iterations, processors view all names $u_i \in G_{j''' < j}$ from previous groups as contended.

Lemma 4.4. *In any execution, at most* $\frac{n}{2^{j-1}}$ *processors ever contend for names from groups* $G_{j' \geq j}$.

Lemma 4.5. *For any fixed* j*, the total number of clean*(j) *iterations is larger than or equal to* $\alpha n + \frac{n}{2^{j-1}}$ *with probability at most* $e^{-\frac{\alpha n}{16}}$ *for all* $\alpha \geq \frac{1}{2^{j-5}}$.

Proof. Fix some j. Consider a time t when the first *dirty*(j) iteration is completed. At time t, there exists an i such that $u_i \in G_{j'' > j}$ and a quorum of processors view $Contended[i] = \mathsf{true}$, so all iterations that start later will set $Contended[i] \leftarrow \mathsf{true}$ on line 6. Therefore, any iteration that starts after t must observe $Contended[i] = \mathsf{true}$ on line 8 and by definition cannot be *clean*(j). By Lemma 4.4, at most $\frac{n}{2^{j-1}}$ processors can have active *clean*(j) iterations at time t. The total number of *clean*(j) iterations is thus upper bounded by $\frac{n}{2^{j-1}}$ plus the number of *clean*(j) iterations completed before time t, which we denote as *safe* iterations.[4] Safe iterations all finish before any iteration where a processor contends for a name in $G_{j'' > j}$ is completed. By Lemma 4.4, at most $\frac{n}{2^j}$ different processors can ever contend for names in $G_{j'' > j}$, therefore, αn safe iterations can occur only if in at most $\frac{n}{2^j}$ of them processors choose to contend in $G_{j'' > j}$. Otherwise, some processor would have to complete an iteration where it contended for a name in $G_{j'' > j}$.

In every *clean*(j) iteration, on line 8, any processor p contends for a name in $G_{j' \geq j}$ uniformly at random among non-contended spots in its view V_p. With probability at least $\frac{1}{2}$, p contends for a name from $G_{j'' > j}$, because by definition of *clean*(j), all spots in $G_{j'' > j}$ are non-contended in V_p.

Let us describe the process by considering a random variable $Z \sim \mathrm{B}(\alpha n, \frac{1}{2})$ for $\alpha \geq \frac{1}{2^{j-5}}$, where each success event corresponds to an iteration contending in $G_{j'' > j}$. By the end, the probability of αn iterations with at most $\frac{n}{2^j}$ processors contending in $G_{j'' > j}$ is:

$$\Pr\left[Z \leq \frac{n}{2^j}\right] = \Pr\left[Z \leq \frac{\alpha n}{2}\left(1 - \frac{2^{j-1}\alpha - 1}{2^{j-1}\alpha}\right)\right] \leq \exp\left(-\frac{\alpha n (2^{j-1}\alpha - 1)^2}{2(2^{j-1}\alpha)^2}\right) \leq e^{-\frac{\alpha n}{8}}$$

So far, we have assumed that the set of names belonging to the later groups $G_{j'' > j}$ was fixed, but the the adversary controls the execution. Luckily, what happened before phase j (i.e. the actual names that were acquired from $G_{j''' < j}$) is irrelevant, because all the names from the earlier phases are viewed as contended by all iterations that start in phases $j' \geq j$. Unfortunately, however, the adversary also influences what names belong to group j and to groups $G_{j'' > j}$. There are $\binom{2^{1-j}n}{2^{-j}n}$ different possible choices for names in G_j, and by a union bound, the probability that αn iterations can occur even for one of them is at most:

$$e^{-\frac{\alpha n}{8}} \cdot \binom{2^{1-j}n}{2^{-j}n} \leq e^{-\frac{\alpha n}{8}} \cdot (2e)^{2^{-j}n} \leq e^{-n(2^{-3}\alpha - 2^{1-j})} \leq e^{-\frac{\alpha n}{16}}, \text{ proving the claim.}$$

□

Plugging $\alpha = \beta - \frac{1}{2^{j-1}} \geq \frac{1}{2^{j-5}}$ in the above lemma, we obtain that the total number of *clean*(j) iterations is larger than or equal to βn with probability at most $e^{-\frac{\beta n}{32}}$ for all $\beta \geq \frac{1}{2^{j-6}}$. Let $X_i(clean)$ be the number of processors that ever contend for $u_i \in G_j$ in some *clean*(j) iteration and define $X_i(dirty)$ analogously: as the number of processors that ever contend for $u_i \in G_j$ in some *dirty*(j) iteration. Relying on the above bound on the number of *clean*(j) iterations, we get the following result:

[4] If no *dirty*(j) iteration ever completes, then we call all *clean*(j) iterations safe.

Lemma 4.6. $\mathbb{E}[\sum_{i=1}^{n} X_i(clean)] = O(n)$.

Each iteration where a processor contends for a name $u_i \in G_j$ is by definition either as $clean(j)$, $dirty(j)$ or starts in a phase $j''' < j$. Let us call these $cross(j)$ and denote by $X_i(cross)$ the number of such iterations. We show that in any execution, for each j, any processor participates in at most one $dirty(j)$ and at most one $cross(j)$ iteration. This allows us to prove with some work the following lemma:

Lemma 4.7. $\mathbb{E}[\sum_{i=1}^{n} X_i(dirty)] = O(n)$ *and* $\mathbb{E}[\sum_{i=1}^{n} X_i(cross)] = O(n)$.

The message complexity upper bound then follows by piecing together the previous claims.

Theorem 4.8. *The expected message complexity of our renaming algorithm is* $O(n^2)$.

Proof. We know $X_i = X_i(clean) + X_i(dirty) + X_i(cross)$ and by Lemma 4.6, Lemma 4.7 we get $\mathbb{E}[\sum_i X_i] = O(n)$. Combining with Lemma 4.3 gives the desired result. □

The time complexity upper bound exploits a trade-off between the probability that a processor collides in an iteration (and must continue) and the ratio of available slots which must be assigned during that iteration.

Theorem 4.9. *The time complexity of the the renaming algorithm is* $O(\log^2 n)$.

Proof Sketch. We fix an arbitrary processor p, and upper bound the number of loop iterations it performs during the execution. Let M_i be the set of free slots that p sees when performing its random choice in the ith iteration of the loop, and let $m_i = |M_i|$.

Assuming that p does not complete in iteration i, let $Y_i \subseteq M_i$ be the set of *new* slots that p finds out have become contended at the beginning of iteration $i+1$, and let $y_i = |Y_i|$. We define an iteration as *low-information* if $y_i/m_i < 1/\log m_i$. Intuitively, in a high-information iteration, the processor collides, but at least reduces its random range for choices by a significant factor.

We then focus on a low-information iteration i. We model the interaction between the processor and the adversary as follows: p first makes a random choice among m_i slots. The adversary can then schedule $m_i - 1$ other processors to make choices in M_i. Importantly, we note that the adversary can schedule some processors several times; however, for each such re-scheduling, it must announce the previously chosen slot. A balls-into-bins argument yields that the adversary can obtain at most m_i extra choices for the iteration to stay *low-information*, with probability at least $1 - 1/m_i$.

Recall that the goal of the adversary is to cause a collision with p's random choice r. We reduce this to a balls-into-bins game in which the adversary throws at most $2m_i - 1$ initial balls into a set of $m_i(1 - 1/\log m_i)$ bins, with the goal of hitting a specific bin, corresponding to p's random choice. However, the probability that an arbitrary bin gets hit in this game is at most a constant.

By the above, processor p terminates in each low-information iteration with constant probability. Putting it all together, we obtain that, for $c \geq 4$ constant, after $c \log^2 n$ iterations, any processor p will terminate with probability $1 - 1/n^{c-1}$, either because m_i reaches 1, or because one of its probes was successful in an low-information phase, which occurs with high probability. □

5. COMMUNICATION LOWER BOUND

In this section, we prove that our algorithms are message-optimal by showing a lower bound of expected $\Omega(n^2)$ messages on any algorithm implementing leader election or renaming in an asynchronous message-passing system where $t < n/2$ processors may fail by crashing. In fact, we prove such a lower bound for any object which exports *strongly non-commutative* methods [12].

Definition 5.1. *Given an object O, a method M of this object is* strongly non-commutative *if there exists some state S of O for which an instance m_1 of M executed sequentially by processor p changes the result of an instance m_2 of M executed by processor $q \neq p$, and vice-versa, i.e. m_2 changes the result of m_1 from state S.*

We now give a message complexity lower bound for objects with non-commutative operations.

Theorem 5.2. *Any implementation of an object O with a strongly non-commutative operation M by $k \leq n$ processors guaranteeing termination with at least constant probability $\alpha > 0$ in an asynchronous message-passing system where $t < n/2$ processors may fail by crashing must have worst-case expected message complexity $\Omega(\alpha kn)$.*

Proof. Let A be an algorithm implementing a shared object O with a strongly non-commutative method M, in asynchronous message-passing with $t < n/2$, guaranteeing termination with probability α. We define an adversarial strategy for which we will argue that all the resulting terminating executions (regardless of their probability) must cause $\Omega(kn)$ messages to be sent. This clearly implies our claim. The strategy proceeds as follows.

Assume that each processor is executing an instance of M. The adversary picks a subset S of $k/4$ participants, and places them in a "bubble:" for each such processor q, the adversary suspends all its incoming and outgoing messages in a buffer, until there are at least $n/4$ such messages in the buffer. At this point, the processor is freed from the bubble, and is allowed to take steps synchronously, together with other processors. Processors outside S execute in lock-step, and their messages outside the bubble are delivered in a timely fashion.

Note that this strategy induces a family of executions $\mathcal{E}$, each of which is defined by the set of coin flips made by the processors. We can assume that there exists a time τ after which in all executions in $\mathcal{E}$ with non-zero probability no processors send any more messages. Otherwise, the adversary can always wait for another message that must be sent, then for the next message, and so on, until $\Omega(kn)$ messages.

Let us prove that in each execution $E \in \mathcal{E}$ every processor in the bubble must eventually leave the bubble before returning, which implies $\Omega(kn)$ messages in executions in which all processors return. Towards this goal, we first show that a processor cannot return while still in the bubble. Then we prove that all processors in the bubble are forced to either return while still in the bubble (which cannot happen) or leave the bubble, completing the proof.

For the first part, assume for the sake of contradiction that there exists an execution $E \in \mathcal{E}$ and a processor $p \in S$ that decides in E while still being in the bubble. Practically, this implies that p has returned from its method invocation without receiving any messages, and without any of its messages being received. To obtain a contradiction, we build

two alternate executions E' and E'', both of which are indistinguishable to p, but in which p must return different outputs.

In execution E', we run all processors outside the bubble until one of them returns–this must eventually occur with constant probability, since this execution is indistinguishable to these processors from an execution in which all (at most $k/4 < n/2$) processors in the bubble are initially crashed. We suspend messages sent to the processors inside the bubble. We then run processor p, which flips the same coins as in E (the execution exists as this happens with probability > 0), observes the same emptiness and therefore eventually returns with constant probability, without having received any messages. We deliver p's messages and suspended messages as soon as p decides.

In execution E'', we first run p in isolation, suspending its messages. With probability > 0, p flips the same coins as in E, and must eventually decide with constant probability without having received any messages. We then run all processors outside the bubble in lock-step. One of these processors must eventually return with constant probability, since to these processors, the execution is indistinguishable from an execution in which p (and other processors in the bubble) has crashed initially. We deliver p's messages after this decision. Since both E' and E'' are indistinguishable to p, it has to return the same value in both executions with constant probability. However, this cannot be the case because instances of method M are strongly non-commutative, the two returning instances are not concurrent, and occur in opposite orders in the two executions. This correctness requirement is enforced *deterministically*. Therefore, p must return distinct values in executions E' and E'', which is a contradiction. Hence, p cannot return in E.

To complete the argument, we prove that p has to eventually return or leave the bubble, with probability $\geq \alpha$. We cannot directly require this of the execution prefix E since not all messages by correct processors have been delivered in this prefix. For this, we consider time τ, at which we crash all recipients of messages by p, and all processors that sent messages to p in E. By the definition of the bubble, the number of crashes we need to expend is $< n/4$. Therefore, by definition of τ, there exists a valid execution, in which no more messages will be sent and p must eventually decide with probability $\geq \alpha$. From p's prospective, the current execution in the bubble can be this execution, and if the adversary keeps p in the bubble for long enough, it has to decide with probability $\geq \alpha$. However, from the previous argument, we know that p cannot decide while in the bubble, therefore p has to eventually leave the bubble in order to be able to decide and return.

This shows that a specific processor p must eventually leave the bubble. The final difficulty is in showing that we can apply the same argument to *all* processors in the bubble at the same time without exceeding the failure budget. Notice however that we could apply the following strategy: for each processor q in the bubble, we could fail all senders and recipients of q ($< n/4$), and also all other processors in the bubble ($< n/4$) at time τ. This can be applied without exceeding the failure budget. Since any processor q could be the sole survivor from the bubble to which we have applied the buffering strategy, and since q does not see a difference from an execution in which it has to return, analogously to the previous case, we obtain that each q in the bubble has to eventually leave the bubble with probability $\geq \alpha$.

Therefore, we obtain that at least $\alpha kn/16$ messages have to be exchanged during the execution, which implies the claim. □

It is easy to check that the *elect* procedure of a leader election algorithm and the *rename* procedure of a strong renaming algorithm are both non-commutative. (In the case of renaming, consider $n+1$ distinct processors executing the rename procedure. By the pigeonhole principle, there exists some non-zero probability that two processors choose the same name in solo executions. Therefore, these two operations do not commute, and therefore the *rename* procedure is strongly non-commutative.) We therefore obtain the following corollary.

Corollary 5.3. *Any implementation of leader election or renaming by $k \leq n$ processors which ensures termination with probability at least $\alpha > 0$ in an asynchronous message-passing system where $t < n/2$ processors may fail by crashing must have worst-case expected message complexity $\Omega(\alpha kn)$.*

6. DISCUSSION AND FUTURE WORK

We have given the first sub-logarithmic leader election algorithm against a strong adversary, and asymptotically tight bounds for the message complexity of renaming and leader election. Our results also limit the power of topological lower bound techniques, e.g. [22], when applied to randomized leader election, since these techniques allow processors to communicate using unit-cost broadcasts or snapshots. Our algorithm shows that no bound stronger than $\Omega(\log^* n)$ time is possible using such techniques, unless the cost of information dissemination is explicitly taken into account.

Determining the tight time complexity bounds for leader election remains an intriguing open question. Another interesting research direction would be to apply the tools we developed to obtain time- and message-efficient implementations of other fundamental distributed tasks, such as task allocation or mutual exclusion, and to explore solutions optimizing bit complexity.

7. ACKNOWLEDGMENTS

The authors would like to thank Prof. Nir Shavit for advice and encouragement during this work, and the anonymous reviewers for their very useful suggestions.

8. REFERENCES

[1] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi. Wait-free test-and-set (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms*, WDAG '92, pages 85–94, London, UK, UK, 1992. Springer-Verlag.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[3] D. Alistarh and J. Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, Sept. 2011.

[4] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and R. Guerraoui. Tight bounds for asynchronous renaming. Accepted to JACM, Sept. 2013.

[5] D. Alistarh, J. Aspnes, V. King, and J. Saia. Communication-efficient randomized consensus. In *Distributed Computing*, pages 61–75. Springer, 2014.

[6] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui. Fast Randomized Test-and-Set and Renaming. In *Proceedings of DISC 2010*, Lecture Notes in Computer Science. Springer-Verlag New York, Ms Ingrid Cunningham, 175 Fifth Ave, New York, Ny 10010 Usa, 2010.

[7] D. Alistarh, M. A. Bender, S. Gilbert, and R. Guerraoui. How to allocate tasks asynchronously. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 331–340, 2012.

[8] D. Alistarh, R. Gelashvili, and A. Vladu. How to elect a leader faster than a tournament. Technical report, http://arxiv.org/abs/1411.1001.

[9] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[10] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, July 1990.

[11] H. Attiya and K. Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):20:1–20:26, Nov. 2008.

[12] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *ACM SIGPLAN Notices*, volume 46, pages 487–498. ACM, 2011.

[13] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[14] J. F. Buss, P. C. Kanellakis, P. L. Ragde, and A. A. Shvartsman. Parallel algorithms with processor failures and delays. *J. Algorithms*, 20:45–86, January 1996.

[15] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, Sept. 1965.

[16] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[17] G. Giakkoupis and P. Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 19–28, New York, NY, USA, 2012. ACM.

[18] G. Giakkoupis and P. Woelfel. A tight rmr lower bound for randomized mutual exclusion. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 983–1002, New York, NY, USA, 2012. ACM.

[19] D. Hendler and P. Woelfel. Randomized mutual exclusion in o(log n / log log n) rmrs. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 26–35, New York, NY, USA, 2009. ACM.

[20] D. Hendler and P. Woelfel. Adaptive randomized mutual exclusion in sub-logarithmic expected time. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 141–150, New York, NY, USA, 2010. ACM.

[21] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[22] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of The ACM*, 46:858–923, 1999.

[23] P. C. Kanellakis and A. A. Shvartsman. Efficient parallel algorithms can be made robust. *Distrib. Comput.*, 5(4):201–217, Apr. 1992.

[24] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[25] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[26] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[27] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 351–362. Society for Industrial and Applied Mathematics, 1991.