Leader Election and Renaming
with Optimal Message Complexity

by

Rati Gelashvili

B.S., Swiss Federal Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Author .................................................................
Department of Electrical Engineering and Computer Science
May 21, 2014

Certified by ..............................................................
Nir Shavit
Professor of Computer Science
Thesis Supervisor

Accepted by ..........................................................
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students
Leader Election and Renaming

with Optimal Message Complexity

by

Rati Gelashvili

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Asynchronous message-passing system is a standard distributed model, where $n$ processors communicate over unreliable channels, controlled by a strong adaptive adversary. The asynchronous nature of the system and the fact that $t < n/2$ processors may fail by crashing are the great obstacles for designing efficient algorithms.

Leader election (test-and-set) and renaming are two fundamental distributed tasks. We prove that both tasks can be solved using expected $O(n^2)$ messages—the same asymptotic complexity as a single all-to-all broadcast—and that this message complexity is in fact optimal.

Thesis Supervisor: Nir Shavit
Title: Professor of Computer Science
Acknowledgments

I would like to thank Professor Nir Shavit for supervision, incredible support and for introducing me to the exciting field of concurrent algorithms. It is my privilege and pleasure to be a student of Nir.

This thesis is based on a join work with Dan Alistarh and Adrian Vladu. I am very thankful to both Dan and Adrian for our enjoyable and productive collaboration.

I would like to thank my family and friends for their support throughout, especially Guranda Darchidze and Ilya Razenshteyn for always being there for me.

Finally, I am very thankful to Akamai and MIT Department of Computer Science and Electrical Engineering for providing the funding for my education and research during the last two years.
## Contents

1 Introduction ...................................................... 11
   1.1 Related Work ........................................... 15
   1.2 Preliminaries ............................................ 16

2 Leader Election .................................................. 19
   2.1 Algorithm ............................................... 19
   2.2 Analysis ................................................ 21

3 Renaming ........................................................ 27
   3.1 Algorithm ............................................... 27
   3.2 Analysis ................................................ 28

4 Message Complexity Lower Bounds ............................. 47

5 Discussion and Future Work .................................. 51

A Test-And-Set for Two Processors ............................. 53
   A.1 MiniTAS algorithm ...................................... 53
   A.2 Analysis ................................................ 55

B Adaptive Test-and-Set ......................................... 59
   B.1 AdaptiveTAS Algorithm ................................. 59
   B.2 Analysis ................................................ 61
List of Figures

2-1 Pseudocode of the PoisonPill Algorithm for $n$ processors. . . . . . . 20

3-1 Pseudocode of the renaming algorithm for $n$ processors. . . . . . . 27

A-1 Pseudocode of the MiniTAS algorithm for two processors. . . . . . . 54

B-1 Pseudocode of the AdaptiveTAS algorithm for $n$ processors. . . . . 60
Chapter 1

Introduction

The theory of distributed computing centers around a set of fundamental problems, also known as tasks. This set includes consensus (agreement) \[ \text{LSP82, PSL80} \], leader election (test-and-set) \[ \text{AGTV92} \], mutual exclusion \[ \text{Dij65} \], renaming \[ \text{ABND}^+90 \], and task allocation (do-all) \[ \text{KS92} \]. These tasks are usually considered in two classic models for distributed computation: asynchronous shared-memory and asynchronous message-passing \[ \text{Lyn97} \].

Some of the most celebrated results of the field are impossibilities, which limit the power of deterministic distributed computation \[ \text{FLP85, HS99} \]. Fortunately, relaxing the task specifications to allow for randomization (and in particular, probabilistic termination) has proved to be a very useful tool for circumventing fundamental impossibilities, and for obtaining efficient algorithms. Starting with seminal work by Ben-Or \[ \text{BO83} \], a tremendous amount of research effort has been dedicated to upper and lower bounds on the complexity of basic tasks in the two fundamental distributed models. In particular, in the asynchronous shared-memory model, (almost) tight complexity bounds are known for randomized implementations of fundamental tasks such as consensus \[ \text{AC08} \], mutual exclusion \[ \text{HW09, HW10, GW12b} \], renaming \[ \text{AACH}^+13 \], and task allocation \[ \text{BKRS96, ABGG12} \].

Somewhat surprisingly, much less is known about the complexity of randomized distributed tasks in the asynchronous message-passing model.\footnote{While simulations between the two models exist \[ \text{ABND95} \], their complexity overhead is linear} In this setting, a set
of $n$ processors communicate via point-to-point channels. Communication is asynchronous, i.e., messages can be arbitrarily delayed. Further, the system is controlled by a strong (adaptive) adversary, which sees the contents of messages and local state, and can choose to crash $t < n/2$ of the participants at any point during the computation. Two natural complexity metrics in this model are message complexity, i.e., total number of messages sent by the protocol, and round complexity, i.e., number of send-receive steps executed by a processor before returning.

We focus on two fundamental tasks: test-and-set (leader election) and strong (tight) renaming. Test-and-set is the distributed equivalent of a tournament: each process must return either a winner or a loser indication, with the property that exactly one process may return winner. In strong renaming, the $n$ processors start out with names from an unbounded namespace, and must return unique names between 1 and $n$. While the complexity of these tasks in shared-memory continues to be an active research topic, e.g. [AAG+10, AA11, GW12a, AACH+13], their message and round complexity in a message-passing system is not well understood.

We give tight bounds for the message complexity of randomized test-and-set and strong renaming in asynchronous message-passing, against a strong adaptive adversary. We prove that expected $\Theta(n^2)$ messages are both necessary and sufficient for solving these tasks if $t < n/2$ processors may fail by crashing. This is somewhat surprising, given that, asymptotically, this is the same cost as a single all-to-all broadcast.

Our first algorithm, called PoisonPill, solves test-and-set using $O(n^2)$ messages with high probability (w.h.p.) and expected $O(\log \log n)$ communication rounds. The algorithm is organized in communication rounds, whose main goal is to eliminate processors from contention. At the beginning of each round, each processor takes a “poison pill" (moves to a 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 (state Low-Priority) or to take an “antidote" (state High-Priority). Each processor then broadcasts its new state, and checks the states of other processors. Crucially, if

---

2 Asynchrony and the ability to crash more than $n/2 - 1$ participants can lead to partitioning, in which case most tasks cannot be solved.
it has chosen not to take the antidote, and sees any other processor either in state Commit or in state High-Priority, the processor returns loser. Otherwise, it continues to the next round.

Correctness follows from the fact that the above construction deterministically ensures that at least one processor survives each round. The key idea behind the algorithm’s efficiency is that, roughly, the only way the adversary could cause a large number of processors to continue is by scheduling all processors which choose to drop before any processor which flips to continue. However, in order to actually examine a processor’s random choice, the adversary must first allow it to take the poison pill (Commit state). Crucially, any other processor observing this Commit state automatically drops out. We essentially prove that, in our scheme, the adaptive adversary can do no more than to allow processors to execute each round in sequence, hoping that the first processor which chooses to continue comes as late as possible in the sequence.

We set up the coin biases such that, within $O(\log \log n)$ rounds, the expected number of processors left in contention is $O(\text{polylog } n)$. Processors surviving after these rounds execute a backup test-and-set based on the RatRace shared-memory algorithm [AAG+10], which will guarantee a single winner after another $O(\log \log n)$ expected rounds. The structure of our protocol is similar to the shared-memory sifting algorithm of [AA11], which assumes a weak (oblivious) adversary.\(^3\) The key difference is the poison pill idea, which allows us to consider a much stronger adversarial model.

Our second algorithm assigns distinct names from 1 to $n$ to the $n$ processors using $O(n^2)$ messages and $O(\log^2 n)$ communication rounds in expectation. The idea behind the algorithm is simple: in each round, each process picks a name at random from the ones it sees as available and communicates its choice; collisions are solved by associating each name with a test-and-set object. This is a natural strategy for renaming, and variants of it have been used previously in shared-memory [AAG+10]. Analyzing the complexity of this simple strategy turns out to be quite complex, and

\(^3\)An oblivious adversary fixes its strategy in advance of the execution, as it cannot observe the random coin flips.
our main technical contribution. Since the processors’ views of available slots may not be coherent, and the strong adversary can manipulate the schedule to maximize the number of collisions on specific names.

We circumvent these obstacles by carefully ordering the processors’ trials based on the set of slots they perceive as available. This allows us to bound the number of collisions during an execution and the number of iterations wasted because of inconsistent views. In turn, these imply a bound on the messages sent and received by the protocol, and on its round complexity.

We match the message complexity of our algorithms with a lower bound on the expected number of messages a test-and-set algorithm must send if it tolerates $t < n/2$ crashes. The idea behind the bound is intuitively simple: no processor should be able to decide if it does not receive any message. Since the adversary can fail up to $t$ processors, it should be able to force a processor to either send or receive $t + 1$ messages. However, this intuition is not trivial to formalize, since groups of processors could employ complex 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 via an 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, which proves the lower bound. A similar argument yields an $\Omega(n^2)$ lower bound for renaming, and in fact for any object with non-commutative operations.

In sum, our results give the first tight bounds on the message complexity of test-and-set and renaming in asynchronous message-passing. Messages in the PoisonPill protocol use $O(\log n)$ bits, while messages in the renaming algorithm may need $\Theta(n)$ bits in the worst case. The expected round complexity of PoisonPill is $O(\log \log n)$, which is exponentially lower than that of its shared-memory strong adversary counterparts [AGTV92, AAG+10]. Our algorithms leverage the broadcast primitive to reduce the number of local computation steps and the contention during the execution. At the same time, the message cost of both algorithms is dominated by a constant number of all-to-all broadcasts, which is asymptotically optimal.


1.1 Related Work

We focus on previous work on the complexity of randomized test-and-set and renaming. To the best of our knowledge, message complexity of these problems in an asynchronous setting against a strong adversary has not been considered before. Since simulations between the two models exist \[ABND95\], one option would be to simply emulate efficient shared-memory solutions in message-passing. However, upon close inspection, one notices that simulating the fastest known shared-memory algorithm for test-and-set \[AGTV92, AAG^{+10}\] would cost $\Theta(n^2 \log n)$ messages and $\Theta(\log n)$ communication rounds. The message cost of simulating the best known strong renaming algorithm \[AACH^{+13}\] is also $\Omega(n^2 \log n)$. Thus, the resulting protocols would be sub-optimal. Moreover, in the case of test-and-set, we show that round complexity can be reduced exponentially by leveraging the broadcast mechanism. Recent work considered the complexity of strong renaming in synchronous message-passing \[GKW13\], giving push-pull algorithm with round complexity $O(\log^2 n)$.

Our algorithms build upon some asynchronous shared-memory tools. In particular, our test-and-set protocol has a similar structure to the sifting procedure introduced in \[AA11\]. (On the other hand, the poison pill technique allows us to tackle a stronger adversarial model while ensuring optimal message complexity.) Our renaming strategy is similar to the shared-memory algorithm of \[AAG^{+10}\]. However, the setting is different, and the analysis technique we develop is more complex, yielding a tight message complexity upper bound.

Concurrent work \[AAKS14\] considered the message complexity of randomized consensus in the same model, presenting an algorithm using $O(n^2 \log^2 n)$ message complexity, and $O(n \log^3 n)$ round complexity, employing fundamentally different techniques.
1.2 Preliminaries

Model. We consider the standard message-passing system model [ABND95]. In this model, \( n \) processors communicate with each other by sending messages through channels. We assume that there is one channel from each processor to every other processor and that the channels form \( i \) to \( j \) and from \( j \) to \( i \) are completely independent. Messages can be arbitrarily delayed by a channel, but do not get corrupted.

Computations are modeled as sequences of steps of the processors. At each step of a processor, first, some messages from incoming channels may be delivered to it and then, unless the processor is faulty, it can perform local computations and send new messages. More formally, a processor is non-faulty, if it is allowed steps to perform local computations and send messages infinitely often and if all messages it sends are eventually delivered. Every message delivery corresponds to a step and note that the messages are also delivered to faulty processors, they just stop performing computations and their messages may be dropped. At most \( t \) processors can be faulty. We consider algorithms that are resilient to at most \( \lceil n/2 \rceil - 1 \) processor failures, i.e. when more than half of the processors are non-faulty, they all eventually terminate.

The Communicate Primitive. Our algorithms use a procedure called communicate, defined in [ABND95] as a building block for asynchronous communication. In essence, calling \( \text{communicate}(m) \) with a message \( m \), involves sending \( m \) to all \( n \) processors and waiting for at least \( \lceil n/2 \rceil + 1 \) acknowledgments before doing anything else. Procedure communicate can be viewed as a resilient broadcast mechanism to mitigate the potential problems caused by delivery delays and faulty processors. The procedure is called with messages of the form \((\text{propagate},v_i)\) or \((\text{collect},v)\). For the first message type, the receiving processor \( j \) updates its view and simply acknowledges by sending back an \( \text{ACK} \) message. In the second case, the acknowledgement is a pair \((\text{ACK},v_j)\) containing the view of the variable for the receiving process.

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, this adversary observes the complete state of the system, including the outcomes of random coin flips performed by the processors, and decides what happens in the future accordingly in an adaptive manner. The goal of the adversary is to maximize the complexity measures defined below.

**Complexity Measures.** We consider two worst-case complexity measures against the adaptive adversary. First is the *message complexity* defined as the expected total number of messages sent by all processors and the second measure is the *round complexity*, which is the maximum expected number of *communicate* calls by some processor. Clearly, our goal is to minimize these complexity measures in the strong adversarial setting.

**Problem Statements.** In the *test-and-set* problem, each processor may return either *WIN* or *LOSE*. A correct algorithm ensures that every (correct) processor returns (*termination*), and that only one processor may return *WIN* (*unique winner*). Also, no processor may lose before the eventual winner starts its execution. *Strong (tight) renaming* requires each of the $n$ participants to eventually return a name between 1 and $n$ (*termination*), and each name must be *unique*. 
Chapter 2

Leader Election

2.1 Algorithm

In this section, we present an algorithm for leader election which terminates in $O(\log \log n)$ communication rounds, using $O(n^2)$ messages, w.h.p., against a strong adaptive adversary. The algorithm is based on two layers. The main layer is designed to reduce the number of processors which can still win to a polylogarithmic number. The surviving processors will access the second layer, which is a linearizable Test-and-Set algorithm, with message complexity $O(kn \log k)$ and round complexity $O(\log k)$ for $k$ participants. The construction of the second layer, called AdaptiveTAS, is described in Appendix B.

The main algorithm, PoisonPill, is specified in Figure 2-1 from the point of view of a participating process. All $n$ processors, irrespective of whether they participate in PoisonPill, react to received messages by replying with acknowledgments according to the communicate procedure. The algorithm receives the unique identifier of a participating processor as an input. The goal is to ensure that the outputs of the processors are linearizable in such a way that the first linearized return value is WIN and every other return value is LOSE. In the following, we call a quorum any set of more than half of the processors.

Variable door stored by the processors implements a simple doorway mechanism.
Input: Unique identifier \( i \) of the participating processor  

Output: WIN or LOSE (result of the Test-and-Set)  

Local variables:  
- Status[\( \log \log n \)]\([n] = \{\top\}; \]
- Views[\([n]\)];  
- int door = 0, \( n' \), round, coin;  
- int Doors[\([n]\)];

1. procedure PoisoonPill\((i)\)
2. \( \text{Doors} \leftarrow \text{communicate(} \text{collect, door} \text{)} \)
3. if \( \exists j : \text{Doors}[j] = 1 \) then return \( \text{LOSE} \)
4. door \( \leftarrow 1 \)
5. \( \text{communicate(} \text{propagate, door} \text{)} \)
   /* Late processor may see a closed door and lose */
6. \( n' \leftarrow n \)
7. for \( \text{round} \leftarrow 1 \) to \( \log \log n \) do
8. \( \text{Status}[\text{round}][i] \leftarrow \text{Commit} \)
9. \( \text{communicate(} \text{propagate, Status[} \text{round}][i] \text{)} \)
   /* Processor committed to a coin flip */
10. \( \text{coin} \leftarrow \text{random(} 1 \text{ with probability } 1/\sqrt{n'}, 0 \text{ otherwise) } \)
11. if \( \text{coin} = 0 \) then \( \text{Status[} \text{round}][i] \leftarrow \text{Low-Pri} \)
12. else \( \text{Status[} \text{round}][i] \leftarrow \text{High-Pri} \)
13. \( \text{communicate(} \text{propagate, Status[} \text{round}][i] \text{)} \)
   /* Priority propagated to more than half of the processors */
14. \( \text{Views} \leftarrow \text{communicate(} \text{collect, Status[} \text{round}] \text{)} \)
15. if \( \text{Status[} \text{round}][i] = \text{Low-Pri} \) then
   16. if \( \exists j : (\exists k : \text{Views}[k][j] \in \{\text{Commit, High-Pri}\} \text{ and } \forall k' : \text{Views}[k'][j] \neq \text{Low-Pri}) \) then
      return \( \text{LOSE} \)
   17. /* Processor with low priority may get eliminated */
18. \( n' \leftarrow 3 \log n \cdot \sqrt{n'} \)
19. \( \text{round} \leftarrow \text{round} + 1 \)
20. return AdaptiveTAS\((i)\)

Figure 2-1: Pseudocode of the PoisonPill Algorithm for \( n \) processors.

A value of 0 corresponds to the door being open and a value of 1 corresponds to the door being closed. Each participating processor \( p \) starts by collecting the views of \( door \) from more than half of the processors at line 2. If a closed door is reported, \( p \) is too late and automatically returns \( \text{LOSE} \). The door is closed by processors at line 4, and this information is then propagated to a quorum.

After passing through the doorway, a processor participates in at most \( \log \log n \) sifting rounds. The idea of each sifting round is that some (but not all) processors
drop out and return *LOSE*. The survivors at line [20] participate in AdaptiveTAS, an algorithm for which the complexity depends on the number of participants. Therefore, more sifted processors means better overall complexity.

In each sifting round, processors announce they are about to flip a random coin (lines 8-9), then obtain a priority based on the outcome of this biased coin flip. Each process then propagates its priority information to a quorum (line 13). Next, it collects the views of current round statuses from a quorum. A process $p$ is sifted and returns *LOSE* at line 17 if both of the following occur: (1) the process $p$ has low priority, and (2), it observes another process $q$ that does not have a low priority in any of the views, but $q$ has either high priority or is committed to flipping a coin (state Commit) in at least one of the reported views. If the processor survives all the sifting rounds, it executes the AdaptiveTAS layer, returning the corresponding result.

### 2.2 Analysis

This section contains the proofs of correctness for PoisonPill and its complexity analysis. We begin by stating the properties of the backup layer.

**Theorem B.2.3.** AdaptiveTAS is a linearizable Test-and-Set algorithm, resilient to $\lceil n/2 \rceil - 1$ process faults. If $k$ processors execute AdaptiveTAS, then the message complexity is $O(kn \log k)$ and its round complexity is $O(\log k)$.

This construction and its correctness proof can be found in Appendix B. One key observation for the correctness of PoisonPill is that not all processors may lose.

**Lemma 2.2.1.** If all processors participating in PoisonPill return, at least one processor returns *WIN*.

**Proof.** Any processor that reaches line [20] of the PoisonPill algorithm participates in the AdaptiveTAS, which by Theorem B.2.3 is a linearizable Test-and-Set algorithm. Therefore, if at least one processor reaches line 20, AdaptiveTAS will have a winner, and that processor will return *WIN*.  

21
Assume for contradiction that all processors that participate in PoisonPill return \textit{LOSE}. By the above observation, this can only happen if each processor returns either at line 3 or at line 17. However, not all processors can return \textit{LOSE} at line 3 as in that case the door would never be closed at line 4. Thus, all processor views would be $\textit{door} = 0$, and no processor would actually be able to return at line 3.

It follows that at least one processor returns \textit{LOSE} at line 17 (in some sifting round). Let $r$ be the largest sifting round in which some processor returns at line 17. Since no processors return in later rounds nor at line 20, all processors that participate in round $r$ actually return \textit{LOSE} at line 17 in round $r$. This implies that all processors must have a low priority in round $r$ as a processor with high priority does not return at line 17.

Processors propagate their low priority information in round $r$ to a quorum by calling the \texttt{communicate} procedure at line 13. Let $i$ be the last processor that completes this \texttt{communicate} call. At this point, all information is already propagated to a quorum, so for every processor $j$ that participates in round $r$, more than half of the processors have a view $\textit{Status[round][j]} = \textit{Low-Pri}$.

Therefore, when processor $i$ proceeds to the line 14 and collects the $\textit{Status[round]}$ arrays from more than half of the processors, 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 in this case, given the conditions for sifting, processor $i$ will not return at line 17 in round $r$. This contradiction completes the proof.

We then show that \texttt{PoisonPill} is a correct linearizable algorithm, resilient to $\lceil n/2 \rceil - 1$ faults.

\textbf{Lemma 2.2.2.} At most one processor can return \texttt{WIN} in \texttt{PoisonPill}.

\textit{Proof.} Observe that for a processor to return \texttt{WIN} in \texttt{PoisonPill}, it must return \texttt{WIN} in \texttt{AdaptiveTAS} at line 20. However, since \texttt{AdaptiveTAS} is a correct Test-and-Set algorithm by \textbf{Theorem B.2.3}, there can be at most one such processor. 

\hfill $\square$
Lemma 2.2.3. PoisonPill algorithm is resilient to \([n/2] – 1\) processor faults.

Proof. We need to show that if less than half of the processors fail, PoisonPill calls of all non-faulty processors will eventually terminate. In this conditions, every \texttt{communicate} call by a non-faulty processor eventually returns. Also, non-faulty processors are given infinitely many computation steps to perform necessary local computations. Therefore, every non-faulty processor eventually returns from the PoisonPill call, because, as described in Figure 2-1, there are only finitely many communication and computation steps, and an invocation of AdaptiveTAS algorithm, which is fault resilient by Theorem B.2.3.

Lemma 2.2.4. PoisonPill is a linearizable Test-and-Set algorithm.

Proof. Given an execution \(\alpha\), where some of the processors fail and not return, consider an extended execution \(\alpha':\) it starts with \(\alpha\), but then the failed processors are resurrected one-by-one and run independently until they return (we only consider executions where more than half of the processors are non-faulty, thus by Lemma 2.2.3, these processors do return). Hence, \(\alpha'\) is an execution where all processors return, and the execution intervals of PoisonPill calls are contained in the corresponding intervals in execution \(\alpha\). Therefore, we can prove linearizability in execution \(\alpha'\) since the linearization points can be carried over to \(\alpha\).

By Lemma 2.2.2 and Lemma 2.2.1 there is exactly one processor that returns \texttt{WIN} in \(\alpha'\). Let us linearize its operation to the invocation point \(P\). We claim that every remaining call (returning \texttt{LOSE} in \(\alpha'\)) can be linearized after \(P\). Assume contrary, then there has to be a call that starts and finishes before the invocation of a call that returns \texttt{WIN}. By definition, the earlier call either closes the door or observes a closed door. Therefore, the later call will observe a closed door at line 2 and will return \texttt{LOSE} at line 3 instead of returning \texttt{WIN}, contradicting our assumption.

Next, we focus on performance. Each round \(r\) has a corresponding value \(n'\), which determines the probability bias on line 10. For the first round, \(n' = n\) and it is updated as \(n' = 3 \log n \cdot \sqrt{n'}\) before every subsequent round. The following lemma gives the rationale behind this choice.
Lemma 2.2.5. Consider some round \( r \) of the PoisonPill algorithm (\( 1 \leq r \leq \log \log n \)), and its corresponding \( n' \) value (used on line 10 in round \( r \)). If at most \( n' \) processors ever enter sifting round \( r \), then, with probability at least \( \frac{n-2}{n} \), at most \( 3 \log n \cdot \sqrt{n'} \) processors will ever enter round \( r + 1 \) (or AdaptiveTAS, if \( r \) is the last round).

Proof. Consider the random coin flips on line 10 in round \( r \). Let us highlight the event when the first processor, say processor \( i \), flips value 1 and argue that no other processor \( j \) that subsequently (or simultaneously) flips value 0 can survive round \( r \). 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 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. Based on this information, processor \( j \) will have to return LOSE at line 17.

Because of the above argument, processors survive \( r \)-th sifting round either if they flip 1 and get high priority, or if they flip 0 strictly before any other processor flips 1.

At most \( n' \) processors entered the \( r \)-th round and flipped a random coin at line 10. Each processor independently flips a biased coin and hence, the number of processors that flip 1 is at most the number of 1’s in \( n' \) Bernoulli trials with success probability \( 1/\sqrt{n'} \). Let us denote this random variable by \( X \) and let \( \mu = E[X] \). Observe that \( \mu = \sqrt{n'} \geq n^{\frac{1}{2\log \log n}} > 1 \). Using a standard Chernoff Bound (e.g. [MU05, Theorem 4.4]) for sufficiently large \( n \), we have:

\[
\Pr[X > \log n \cdot \mu] < \left( \frac{e^{\log n - 1}}{\log n^{\log n}} \right)^{\mu} < \frac{1}{n}
\]  

meaning that the probability of more than \( \log n \cdot \sqrt{n'} \) processors flipping 1 is very low.

Let us now consider processors that flip 0 before the first 1 is flipped. For this, let us argue that with high probability one of the first \( 2 \log n \cdot \sqrt{n'} \) coin flips will result in 1. Consider the random variable \( Y \), counting the number of successes in \( 2 \log n \cdot \sqrt{n'} \) Bernoulli trials. Then \( E[Y] = 2 \log n \) and again applying a Chernoff Bound for
sufficiently large $n$ gives $\Pr[Y < 1] < 1/n$. Combining this bound with (2.2.1), the probability of more than $3 \log n \cdot \sqrt{n'}$ processors surviving round $r$ is at most $2/n$. □

The proof of our main claim follows from the previous arguments. To get the desired complexity guarantees, we iteratively consider all $\log \log n$ sifting rounds.

**Theorem 2.2.6.** PoisonPill is a linearizable Test-and-Set algorithm, resilient to $\lceil n/2 \rceil - 1$ processor faults. It has round complexity $O(\log \log n)$ and message complexity $O(n^2)$.

**Proof.** By Lemma 2.2.3, for at most $\lceil n/2 \rceil - 1$ processor faults, PoisonPill calls of all non-faulty processors eventually terminate and by Lemma 2.2.4, PoisonPill is a linearizable Test-and-Set algorithm.

Each processor makes at most $O(\log \log n)$ calls (constantly many per round) to communicate in PoisonPill before possibly executing the AdaptiveTAS algorithm. Using Lemma 2.2.5 and the Union Bound for all $\log \log n$ rounds, with probability at least $\frac{n-2 \log \log n}{n}$, at most $(3 \log n)^{1+1/2+1/4+\ldots} \cdot \frac{\log \log n}{n \log \log n} = O(\log^2 n)$ processors enter AdaptiveTAS. Otherwise, there can be at most $n$ participants. The expected round complexity is hence $O(\log \log n + \frac{\log \log n \cdot \log n}{n}) = O(\log \log n)$.

Analogously, with probability at least $\frac{n-2 \log \log n}{n}$, the number of messages will be at most $O(n(\log^2 n \cdot (n^{1/2} + n^{1/4} + \ldots + 1))) = O(n^2)$. Otherwise, it can be at most $O(n^2)$ messages exchanged in each of the $\log \log n$ sifting rounds, and at most $n^2 \log n$ messages in the final AdaptiveTAS. Therefore, the expected message complexity is: $O(n^2 + \frac{n^2 \log n \cdot \log \log n}{n}) = O(n^2)$. □
Chapter 3
Renaming

3.1 Algorithm

**Input**: Unique identifier \( i \) from a large namespace

**Output**: \( \text{int } name \in [n] \)

**Local variables**:
- \( \text{bool } Contended[n] = \{\text{false}\} \)
- \( \text{int } Views[n][n] \)
- \( \text{int } coin, spot, outcome \)

```plaintext
procedure getName\( \langle i \rangle \)
while true do
   Views ← communicate(collect, Contended)
   for \( j \leftarrow 1 \) to \( n \) do
      if \( \exists k : Views[k][j] = \text{true} \) then
         Contended\[j\] ← true
      /* Processor marks names that became contended */
   communicate(propagate, \{Contended[\( j \) | Contended[\( j \)] = true\}])
   spot ← random(\( j \) | Contended[\( j \)] = false)
   Contended[spot] ← true
   outcome ← AdaptiveTAS\( _{\text{spot}} \)(i)
   /* Processor contends for a new name, propagates contention information */
   communicate(propagate, Contended[spot])
   if outcome = \text{WIN} then
      return \( \text{spot} \)
   /* Exactly one contender wins a name */
```

Figure 3-1: Pseudocode of the renaming algorithm for \( n \) processors.
The algorithm is described in Figure 3-1. There is a separate AdaptiveTAS protocol for each name; a processor claims a name by winning the associated AdaptiveTAS. Each processor repeatedly chooses a new name and contends for it by participating in the corresponding Test-and-Set protocol, 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.

3.2 Analysis

We begin by proving the correctness of the algorithm.

**Lemma 3.2.1.** The renaming algorithm in Figure 3-1 ensures termination in the presence of at most \(\lceil n/2 \rceil - 1\) processor faults and no two processors return the same name from the `getName` call.

*Proof.* Assume that less than half of the processors are faulty. Because the algorithm only uses the `communicate` procedure and the AdaptiveTAS algorithm, which is resilient by Theorem B.2.3, non-faulty processors will always make progress (i.e. they will keep contending for new names). There are at most \(n\) names, and processors never contend for the same name twice (since the first time they set `Contended ← true`, which prohibits contending in the future). Thus, each non-faulty processor eventually terminates.

A processor that returns some name \(u\) from a `getName` call has to be the winner of the AdaptiveTAS\(_u\) protocol. On the other hand, according to Theorem B.2.3, AdaptiveTAS\(_u\) can not have more than one winner.

We then focus on message complexity. We begin by introducing some notation. For an arbitrary execution, and for every name \(u\), consider the first time when more than half of processors have `Contended[u] = 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 3.2.2.** In any execution, if a processor views \( \text{Contended}[i] = \text{true} \) in some while loop iteration, and then, in some subsequent iteration at line 28 the same processor views \( \text{Contended}[j] = \text{false} \), \( i \prec j \) has to hold.

**Proof.** Clearly, \( j \neq i \) because contention information never disappears from a processor’s view. In the earlier iteration, the processor propagates \( \text{Contended}[i] = \text{true} \) to more than half of the processors at line 27 or 31. During the subsequent iteration, at line 23, the processor collects information and does not set \( \text{Contended}[j] \) to \text{true} before reaching line 28. Thus, more than half of the processors view \( \text{Contended}[j] = \text{false} \) at some intermediate time point. Therefore, a quorum of processors views \( \text{Contended}[i] = \text{true} \) strictly earlier than \( \text{Contended}[j] = \text{true} \), and by definition \( i \prec j \) has to hold.

Let \( X_i \) be a random variable, denoting the number of processors that ever contend in a Test-and-Set for the name \( u_i \). The following holds:

**Lemma 3.2.3.** The message complexity of our renaming algorithm is \( O(n \cdot E[\sum_{i=1}^{n} X_i^2]) \).

**Proof.** Let \( Y_j \) be the number of loop iterations executed by a processor \( j \). Then \( \sum_i X_i = \sum_j Y_j \), because every iteration involves one processor contending at a single name spot, and a processor can not contend for a name twice. Each iteration involves two \text{communicate} calls, costing \( 3n \) messages and the number of messages sent in \text{AdaptiveTAS} when a process contends for a name is \( O(\sum_{i:X_i>0} n \cdot X_i \log X_i) \). The
total message complexity is thus the expectation of:

\[
O \left( \sum_{i : X_i > 0} n \cdot X_i \log X_i \right) + \sum_j 3n \cdot Y_j = O \left( \sum_{i : X_i > 0} n \cdot X_i (\log X_i + 3) \right)
\]

\[
\leq O \left( \sum_i n \cdot X_i (X_i + 3) \right)
\]

\[
\leq O \left( n \cdot \sum_i X_i^2 \right)
\]
as desired. \qed

Let us partition names \( \{u_i\} \) into \( \log n \) groups, where the first group \( G_1 \) contains the first \( n/2 \) names, the second group \( G_2 \) contains the next \( n/4 \) names, etc. We will use notations \( G_{j'} \geq j, G_{j''} > j \) and \( G_{j''' < j} \) to denote 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 \( \log n \) 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 \( \text{Contended}[u_i] = \text{true} \) (the way \( u_i \) are 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 \( \text{Contended}[u_i] = \text{true} \). 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 22 and reaches line 23. Let \( V_p \) be \( p \)'s view of the \( \text{Contended} \) array right before picking a spot at line 28 in the given iteration:

- We say that an iteration is \( \text{clean}(j) \), 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 \( \text{dirty}(j) \), 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 3.2.4.** In any execution, at most $\frac{n}{2^{j-1}}$ processors ever contend for names from groups $G_{j'\geq j}$.

**Proof.** If no name from $G_{j'\geq j}$ is ever contended, then the statement is trivially true. On the other hand, if some name from $G_{j'\geq j}$ is ever contended, then by our ordering so are all names from the former groups. (Otherwise, an uncontended name from earlier groups would have time $\infty$ and the least priority and would be sorted later). There are $n - \frac{n}{2^{j-1}}$ names in earlier groups. Since they all are contended at some time, there are $n - \frac{n}{2^{j-1}}$ processors that can be linearized to win the corresponding AdaptiveTAS protocols and the name. Consider one of these processors $p$ and the name $u$ from some earlier group $G_{j''<j}$, that $p$ is bound to win. Processor $p$ will not contend for names after $u$, and it also never contended for a name from $G_{j'\geq j}$ before contending for $u$, because that would have violated Lemma 3.2.2. Thus, none of the $n - \frac{n}{2^{j-1}}$ processors ever contend for a name from $G_{j'\geq j}$, out of $n$ processors in total, completing the argument. \qed

**Lemma 3.2.5.** 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 $e^{-\frac{\alpha n}{2}}$ for all $\alpha \geq \frac{1}{2^{j-1}}$.

**Proof.** Fix some $j$. Consider a time point $t$ when the first dirty($j$) iteration is completed. At this time, there exists an $i$ such that $u_i \in G_{j''>j}$ and more than half of the processors view Contended[$i$] = true, so all iterations that start later will set Contended[$i$] ← true at line 26. Therefore, any iteration that starts after time $t$ will observe Contended[$i$] = true at line 28 and by definition cannot be clean($j$). By Lemma 3.2.4, 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.
Let us now consider only safe iterations, during which, by definition, no iteration is completed in $G_{j'' > j}$, implying that no processor can contend in $G_{j'' > j}$ twice. By Lemma 3.2.4, at most $\frac{n}{2^j}$ different processors can ever contend in $G_{j'' > j}$, therefore, $\alpha n$ safe iterations can occur only if in at most $\frac{n}{2^j}$ of them processors choose to contend in $G_{j'' > j}$.

In every clean($j$) iteration, at line 28, the 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 $Z \sim B(\alpha n, \frac{1}{2})$ for $\alpha \geq \frac{1}{2^{j-\pi}}$, where each success event corresponds to an iteration contending in $G_{j'' > j}$. By the Chernoff Bound, the probability of $\alpha n$ 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 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 $(\frac{2^{1-j} n}{2^{j}})$ different possible choices for names in $G_j$, and by a union bound, the probability that $\alpha n$ iterations can occur even for one of them is at most:

$$e^{-\frac{\alpha n}{8}} \cdot \left( \frac{2^{1-j} n}{2^{j}} \right) \leq e^{-\frac{\alpha n}{8}} \cdot (2e)^{2^{-j} n} \leq e^{-n (2^{-j} \alpha - 2^{1-j})} \leq e^{-\frac{\alpha n}{16}}$$

proving the claim. \hfill \qed

Plugging $\alpha = \beta - \frac{1}{2^{j-\pi}} \geq \frac{1}{2^{j-\pi}}$ in the above lemma, we obtain the following

**Corollary 3.2.6.** The total number of clean($j$) iterations is larger than or equal to
\( \beta n \) with probability at most \( e^{-\frac{\beta n}{32}} \) for all \( \beta \geq \frac{1}{2^{j-6}} \).

Let \( X_i(\text{clean}) \) be the number of processors that ever contend for \( u_i \in G_j \) in some \( \text{clean}(j) \) iteration and define \( X_i(\text{dirty}) \) analogously as the number of processors that ever contend for \( u_i \in G_j \) in some \( \text{dirty}(j) \) iteration. The proof of the following claim is elementary and can for instance be found in [MU05, Example 3.2.1].

**Claim 3.2.7.** Consider a binomial random variable \( Z \sim B(m, q) \). Then, \( \mathbb{E}[Z^2] = m(m-1)q^2 + mq \).

Relying on Corollary 3.2.6 and the above claim, we get the following result:

**Lemma 3.2.8.** \( \mathbb{E}[\sum_{i=1}^{n} X_i(\text{clean})^2] = O(n) \).

**Proof.** Let us equivalently prove that \( \mathbb{E}[X_i(\text{clean})^2] = O(1) \) for any name \( u_i \) in some group \( G_j \), where \( X_i(\text{clean}) \) is defined as the number of \( \text{clean}(j) \) iterations, in which processors contend for a name \( u_i \in G_j \).

By definition, in all \( \text{clean}(j) \) iterations a processor observes all names in \( G_j' \) as uncontended at line 28. Therefore, each time, independent of other iterations, the probability of picking spot \( i \) and contending for the name \( u_i \) is at most \( 2^{j-6} \). Thus, if there are exactly \( \beta n \) of \( \text{clean}(j) \) iterations, \( X_i(\text{clean}) \leq B(\beta n, \frac{2^j}{n}) \). Applying Claim 3.2.7 gives

\[
\mathbb{E}[X_i(\text{clean})^2 \mid u_i \in G_j, \beta n \text{ iterations}] \leq O(\beta^2 n^2 \frac{2^{2j}}{n^2} + \beta n \frac{2^j}{n}) = O(2^{2j} \beta^2 + 2^j \beta) \quad (3.2.1)
\]

for \( \beta n \) clean iterations that started in phase \( j \). The probability that there are exactly \( \beta n \) of \( \text{clean}(j) \) iterations is trivially upper bounded by the probability that there are at least \( \beta n \) \( \text{clean}(j) \) iterations, which by Corollary 3.2.6 is at most \( e^{-\frac{\beta n}{32}} \) for \( \beta \geq \frac{1}{2^{j-6}} \). Therefore:

\[
\mathbb{E}[X_i(\text{clean})^2 \mid u_i \in G_j] \leq 2^{2j} \left( \frac{1}{2^{j-6}} \right)^2 + \frac{2^j}{2^{j-6}} + \sum_{l=\left\lceil \frac{n}{2^{j-6}} \right\rceil}^{\infty} e^{-\frac{l}{32}} \left( 2^{2j} \left( \frac{1}{n} \right)^2 + \frac{2^j l}{n} \right) \quad (3.2.2)
\]

which, after some calculation, is \( O(1) \), completing the proof.
Next, we focus on dirty iterations, and start by proving the following lemma.

**Lemma 3.2.9.** In any execution, any processor participates in at most one dirty($j$) iteration. Furthermore, when a processor completes its dirty($j$) iteration, phase $j$ must already be over.

**Proof.** The first time some processor $p$ participates in a dirty($j$) iteration, by definition, it views $\text{Contended}[i] = \text{true}$ for some $u_i \in G_{j'' > j}$. Therefore, $p$ also propagates $\text{Contended}[i] = \text{true}$ at line 27 in the same iteration. When $p$ starts a subsequent iteration, a quorum of processors know about $u_i \in G_{j'' > j}$ being contended. By the way names in $u$ are sorted, at that point more than half of the processors must already know that each name in $G_j$ is contended, meaning that phase $j$ has ended. Therefore, no subsequent iteration of the processor can be of a dirty($j$) type. \hfill \Box

The proof of next lemma is the most technically involved in this argument.

**Lemma 3.2.10.** $E[\sum_{i=1}^{n_i} X_i(\text{dirty})^2] = \mathcal{O}(n)$.

**Proof.** Recall that $X_i(\text{dirty})$ is the number of processors that ever contend for $u_i \in G_j$ in a dirty($j$) iteration. Let us equivalently fix $j$ and prove that $E[\sum_{u_i \in G_j} X_i(\text{dirty})^2] = \mathcal{O}(\frac{n}{2^{j+1}})$, implying the desired statement by linearity of expectation and telescoping.

We sum up quantities $X_i(\text{dirty})^2$ for the names in $j$-th group, but the adversary controls precisely which names belong to group $G_j$. We will therefore consider all names $u_i \in G_{j' \geq j}$ and sum up quantities $X_{i,j}$: the number of processors that contend for a name $u_i$ in a dirty($j$) iteration.

We start by introducing necessary notations to facilitate the proof:

**Notations:** Let us call all iterations that start in phase $j$ relevant. All dirty($j$) iterations are relevant because they start in phase $j$ by definition. Relevant iterations select (and contend for) spots in $u_i \in G_{j' \geq j}$ (they collect contention information after phase $j$ has started, so all earlier names are viewed as contended). Since we only care about relevant iterations, we will also only ever consider these names $u_i$ and will
sometimes omit writing \( u_i \in G_{j' \geq j} \) for brevity. Using this notation, our goal is to prove that \( \mathbb{E}[\sum_{u_i} X_{i,j}^2] = O(\frac{n^2}{2^{\mathcal{H}_j}}) \).

Consider any execution after the start of phase \( j \), whereby the adversary builds the schedule by delivering some messages and letting the processors of its choice take steps. At any point in the execution, we say that the execution is in configuration \( C \), if the number of processors that picked spot \( u_i \) in their last relevant iteration is \( C[u_i] \). Define the potential at any point in the execution as \( H_C = \sum_{u_i} (\max(C[u_i] - 2, 0))^2 \), where \( C \) is the configuration of the execution at the given point. A spot selection happens in an execution each time some processor picks a name to contend by executing line \( 28 \) of the algorithm in a relevant iteration. Observe that spot selections in execution are precisely the times when the configuration changes. At any point in the execution where the configuration is \( C \), we say that spot (name) \( u_i \) has \( C[u_i] \) processors on it.

In any execution when phase \( j \) starts, the initial configuration has zero processors on every spot, because no spot selection has yet happened. Let us start with this initial configuration and consider spot selections one-by-one, in the order they happen, breaking ties arbitrarily. We update the configuration each time before considering the next spot selection, even for concurrent spot selections. Note that, for concurrent spot selections that lead to some final configuration, we also arrive to the same final configuration but after considering a few intermediate configurations. We call a spot selection by some processor \( p \) meaningful, if \( p \) picks a spot with either zero or two processors on it (that is, spot \( u_i \) with \( C[u_i] \neq 1 \)) in the actual (possibly intermediate) configuration.

During the spot selection by some processor \( p \), let us denote by target spots the set of spots that \( p \) viewed as uncontended and that do not have one processor on them in the actual configuration. By definition, in a meaningful spot selection a processor selects one of its target spots, each with an equal probability, since processors select uniformly at random at line \( 28 \).

Next, let us outline the main challenges and the proof strategy:
The proof strategy: A major difficulty is that the adversary controls scheduling and can strategically influence which relevant iterations of which processors are dirty \( j \). We know by Lemma 3.2.4 that at most \( \frac{n}{2^j - 1} \) processors can have a relevant iteration, implying \( \sum_{\mathcal{U}_i} X_{i,j} \leq \frac{n}{2^j - 1} \). Also, by Lemma 3.2.9 each processor can participate in at most one dirty \( j \) iteration. Furthermore, by the same lemma, the dirty \( j \) iteration of any processor must be the last iteration that the processor undertakes in phase \( j \). This allows us to overcome the challenge by overcounting: we can assume that the last relevant iteration of each of these at most \( \frac{n}{2^j - 1} \) processors is the only dirty \( j \) iteration of the processor. Still, the adversary can end phase \( j \) during different iterations of a given processor, and this way it influences which iteration for each processor will be the last relevant iteration.

Let \( Y_{i,j} \) be the number of processors that contend for name \( u_i \in G_{j' \geq j} \) in their last relevant iteration, such that when phase \( j \) ends, they have already selected the spot \( u_i \) in this last relevant iteration. The adversary controls when phase \( j \) ends, at which time, some processors have ongoing iterations, and in some of the iterations spots have already been selected. If for all spots \( u_i \in G_{j' \geq j} \) we look at the number of processors that selected \( u_i \) in their last spot selection up to the stopping point, we are guaranteed to count all \( Y_{i,j} \) iterations. So, if \( C \) is the final configuration of the execution when phase \( j \) ends, then \( \sum_{\mathcal{U}_i} Y_{i,j}^2 \) is at most \( \sum_{\mathcal{U}_i} C[ u_i ]^2 = O( H_c + \frac{n}{2^j - 1} ) \). We will next get an upper bound of \( O( \frac{n}{2^j - 1} ) \) on the maximum expected value of potential \( H_c \) that the adversary can achieve when it phase \( j \) finishes, (where maximum is taken over all possible adversarial strategies) implying \( \mathbb{E}[ \sum_{\mathcal{U}_i} Y_{i,j}^2 ] = O( \frac{n}{2^j - 1} ) \). We start by the following claim:

Claim 3.2.11. At the time of a spot selection by some processor \( p \), let \( s \) be the number of target spots with at least two processors on them, and let \( a_1, a_2, \ldots, a_s \) be the exact number of processors on these \( s \) spots. Denote the sum by \( S = \sum_{i=1}^{s} a_i \). If \( p \) does a meaningful spot selection, the probability of selecting any fixed one of these \( s \) spots is

---

\(^1\) Counting \( \mathbb{E}[ \sum_{i=1}^{n} X_i(\text{dirty}) ] \) is simple, as it is at most \( \sum_{j=1}^{\log n} \sum_{u_i \in G_{j' \geq j}} X_{i,j} = O(n) \). However, to sum up squares the same approach works only for the last \( \log n/2 \) phases (with less than \( \sqrt{n} \) dirty iterations each).
at most $1/S$. 

which can be used further to prove the following two claims:

**Claim 3.2.12.** The probability of $\alpha n$ meaningful spot selection events occurring in an execution is at most $e^{-\alpha n}$ for $\alpha \geq \frac{1}{27}$. 

**Claim 3.2.13.** Consider any state in any execution where the actual configuration has potential $H$. Then, if the adversary may continue the execution such that at most $m$ meaningful slot selections are performed, and stop anytime, the maximum expected potential the adversary can achieve in final configurations after stopping is at most $H + 3m$.

The proofs of the claims are provided later. The potential of the initial configuration (when phase $j$ starts) is 0. If the adversary runs the execution and performs $\alpha n$ meaningful operations, by **Claim 3.2.13** it can achieve at most $3\alpha n$ expected potential. On the other hand, by **Claim 3.2.12** for $\alpha > \frac{1}{27}$, the probability that the adversary can execute $\alpha n$ meaningful iterations is at most $e^{-\frac{\alpha n}{8}}$ (before phase $j$ ends, or even in the whole remaining execution after phase $j$ starts). Therefore, the maximum expected potential the adversary can achieve is:

$$3 \cdot \frac{n}{2^{j-3}} + \sum_{l=\frac{n}{2^{j-3}}}^{\infty} e^{-\frac{l}{8}}(3l)$$

which after some work, is $O(\frac{n}{2^{j-1}})$. This implies $E[\sum_{u_i \in G_{j' \geq j}} Y_{i,j}^2] = O(\frac{n}{2^{j-1}})$.

Now, let $Z_{i,j}$ be the number of processors that contend for name $u_i \in G_{j' \geq j}$ in their last relevant iteration, but which had not yet selected the spot $u_i$ in the last relevant iteration when phase $j$ ended. Clearly $X_{i,j} = Y_{i,j} + Z_{i,j}$ and $E[\sum_{u_i} X_{i,j}^2] = O(E[\sum_{u_i} Y_{i,j}^2] + E[\sum_{u_i} Z_{i,j}^2])$. We now only need to show that $E[\sum_{u_i} Z_{i,j}^2] = O(\frac{n}{2^{j-1}})$.

By **Lemma 3.2.4**, at most $\frac{n}{2^{j-1}}$ processors contend for names $u_i \in G_{j' \geq j}$, so when phase $j$ ends, all names in $G_j$ should be occupied and at most $\frac{n}{2^{j-1}}$ processors can be executing an iteration where a spot is not yet selected. This is a fixed set of at most $\frac{n}{2^{j-1}}$ iterations, for which the following holds:
Claim 3.2.14. At any point in the execution, consider a fixed set of iterations $I$, in each of which the spot is not yet selected at line 28. Also, consider a fixed set of names $\Gamma$. For any name $u_i \in \Gamma$ let $Z_i$ be the expected number of iterations from $I$ that pick $u$. Then, $\mathbb{E}[\sum_{u_i \in \Gamma} Z_i] = O(\max(|I|, |\Gamma|))$.

Setting $\Gamma = G_{j' \geq j}$, the above claim implies $\mathbb{E}[\sum_{u_i \in G_{j' \geq j}} Z_{i,j}] = O\left(\frac{n^2}{2^j-1}\right)$, completing the proof.

Claim 3.2.11. At the time of a spot selection by some processor $p$, let $s$ be the number of target spots with at least two processors on them, and let $a_1, a_2, \ldots, a_s$ be the exact number of processors on these $s$ spots. Denote the sum by $S = \sum_{i=1}^{s} a_i$. If $p$ does a meaningful spot selection, the probability of selecting any fixed one of these $s$ spots is at most $1/S$.

Proof. There are $\frac{n}{2^j-1}$ names in later groups $G_{j' \geq j}$ and by Lemma 3.2.4 at most $\frac{n}{2^j-1}$ processors ever contend for these names. In the renaming algorithm, after some processor picks the spot, a processor can be linearized to win the Test-and-Set for the name associated with the corresponding spot, and the winner processor never moves to another spot. By a simple pigeon-hole argument, at least $S - s$ names in groups $G_{j' \geq j}$ have never been selected by any processor at line 28 (in neither relevant nor irrelevant iterations) by the time of the meaningful spot selection by processor $p$.

All these $S - s$ spots must have zero processors on them at the time of the meaningful spot selection by processor $p$ and they must have been viewed as uncontended by $p$. Hence, all these $S - s$ empty spots are also target spots, meaning that there are at least $s + (S - s) = S$ target spots in total. Each target spot has an equal probability of being selected at line 28, so if a meaningful spot selection happens, each spot can have a maximum probability of $1/S$ of being selected.

Claim 3.2.12. The probability of $\alpha n$ meaningful spot selection events occurring in an execution is at most $e^{-\frac{\alpha n}{2}}$ for $\alpha \geq \frac{1}{2^j-3}$.

Proof. By Claim 3.2.11, at each meaningful spot selection an empty spot is selected with probability at least $\frac{1}{2}$, because by definition, either an empty spot or a spot
with at least two processors is selected and \( \frac{s}{S} \leq \frac{1}{2} \) (as each \( a_i \geq 2 \)). The adversary continuously lets some processor select spot at line 28, and when it does so, with some probability the selection will turn out meaningful (if the processor picks one of the target spots based on the actual configuration). Given that the selection event ends up meaningful, we know with probability at least \( \frac{1}{2} \) an empty spot is selected, plus the selection is already made and there is no going back for the adversary. However, an empty spot can be selected at most \( \frac{n}{2^{j-1}} \) times, since there are \( \frac{n}{2^{j-1}} \) total spots and a spot never becomes empty in a configuration after being selected once (in the renaming algorithm, after some processor picks the spot, a processor can be linearized to win corresponding Test-and-Set, and the winner never moves to another spot). So, \( \alpha n \) meaningful spot selection events only occur in an execution if an empty spot is selected in at most \( \frac{n}{2^{j-1}} \) of them.

Borrowing part of the analysis from Lemma 3.2.5, we describe the process by a random variable \( Z \sim \text{B(} \alpha n, \frac{1}{2} \text{)} \), where success corresponds to selecting an empty spot. Using Chernoff bound, for \( \alpha \geq \frac{1}{2^{j-3}} \) the probability of \( \alpha n \) meaningful spot selection events occurring in an execution is at most:

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

completing the proof of the claim.

**Claim 3.2.13.** Consider any state in any execution where the actual configuration has potential \( H \). Then, if the adversary may continue the execution such that at most \( m \) meaningful slot selections are performed, and stop anytime, the maximum expected potential the adversary can achieve in final configurations after stopping is at most \( H + 3m \).

**Proof.** The definition of meaningful iterations implies that in any configuration iterations that are not meaningful can only lead to configurations with smaller or equal potential. Therefore, without loss of generality we can assume that there are only meaningful iterations.
Let us prove the statement by induction on $m$. Base case when $m = 0$ is trivial because the adversary stops right away and the expected potential is $H$.

Now assume that the statement is true for at most $m - 1$ meaningful iterations. The adversary starts in a configuration with potential $H$. It can stop right away and achieve potential $H$. Otherwise, consider the first meaningful iteration and let us say that the actual configuration has $s$ target spots with at least two processors on, the exact number of processors on each of these spots being $a_1, a_2, \ldots, a_s$. By [Claim 3.2.11] the probability of selecting the spot with $a_1$ processors on it is at most $1/S$, in which case potential becomes $H + 2a_1 + 1$. From that configuration on, with $m - 1$ more meaningful iterations, the adversary can achieve at most a potential of $H + 2a_1 + 1 + 3(m - 1)$ when it stops. Analogously, with probability at most $1/S$, the adversary can achieve potential at most $H + 2a_1 + 1 + 3(m - 1)$ when it stops. Otherwise, if none of the $s$ spots with at least two processors is picked, the potential in the resulting configuration is $H$, and again by inductive hypothesis, the adversary can achieve final potential at most $H + 3(m - 1)$.

If the adversary stops right away, it achieves a potential $H$, and otherwise expected potential of $H + 3(m - 1) + \sum_i \frac{2a_i + 1}{S} \leq H + 3m$ (more than $H$) when it stops, completing the proof. 

Claim 3.2.14. At any point in the execution, consider a fixed set of iterations $I$, in each of which the spot is not yet selected at line 28. Also, consider a fixed set of names $\Gamma$. For any name $u_i \in \Gamma$ let $Z_i$ be the expected number of iterations from $I$ that pick $u$. Then, $E[\sum_{u_i \in \Gamma} Z_i^2] = O(\max(|I|, |\Gamma|))$.

Proof. We will refer iterations in $I$ as the frozen iterations. Starting from the time point when $I$ is fixed, let us define a configuration $C$, describing for each spot $u_i \in \Gamma$ how many of these iterations have selected $u_i$, denoted by $C[u_i]$. As in [Lemma 3.2.10] define potential to be $H_C = \sum_{u_i \in \Gamma} (\max(C[u_i] - 2, 0))^2$. Initially, all $C[u_i]$ are zero, so the potential of this initial configuration is 0. If $H_f$ is the maximum expected potential that the adversary can achieve after all iterations in $I$ pick spots, then $E[\sum_{u_i \in \Gamma} Z_i^2] = O(|\Gamma|) + O(H_f)$. Now, we need to show that $H_f = O(|I|)$. 

40
Consider spot selections at line 28 in frozen iterations in the order they happen, breaking ties arbitrarily. Among the spots in \(\Gamma\) that are viewed as uncontended by processor \(p\), there will be some, let us say \(s\) of them, that have already been selected at least two frozen iterations. Let the exact number of frozen iterations that have picked these spots be \(a_1, \ldots, a_s\), and let \(S = \sum_{i=1}^{s} a_i\). Each processor can participate in at most one frozen iteration, and by a pigeon-hole argument similar to the one in the proof of [Claim 3.2.11], at least \(S - s\) spots not yet picked in any frozen iteration must be viewed as uncontended by \(p\). There are as many names as processors, and these \(S - s\) spots can be seen as the ones that the \(S - s\) “extra” processors on \(s\) locations would eventually win. Note that these “extra” processors may have already picked these final spots of theirs at the time of spot selection by \(p\), however, they could not have propagated the contention information of the final spot to \(p\): otherwise, since the messages are never reordered in the system, one of the original \(s\) spots would also be viewed as contended, a contradiction.

We conclude that the probability of selecting each spot is at most \(1/S\). Now, observe that [Claim 3.2.13] still holds for this new definition of configurations and potential and implies the desired \(H_f \leq 3|I|\). \qed

Each iteration where a processor contends for a name \(u_i \in G_j\) is by definition either as \(\text{clean}(j)\), \(\text{dirty}(j)\) or starts in a phase \(j'' < j\). Let \(X_i(\text{cross})\) be the number of such iterations. Then, using the same approach as for dirty iterations, we get:

**Lemma 3.2.15.** \(\mathbb{E}[\sum_{i=1}^{n} X_i(\text{cross})^2] = O(n)\).

**Proof.** Any name \(u_i\) in \(G_j\) can be picked only in iterations starting in phases up to \(j\). An iterations that starts in phase \(j\) and picks \(u_i\) is either \(\text{clean}(j)\) or \(\text{dirty}(j)\) and the quantity \(X_i(\text{cross})\) contains only iterations that start in phases \(j'' < j\) and pick \(u_i \in G_j\). These iterations are not finished when (phase \(j - 1\) finishes and) phase \(j\) starts (otherwise \(u_i\) would not be in phase \(G_j\)). Let \(Y_i\) be the number of iterations that have already picked spot \(u_i\) when phase \(j\) starts, and let \(Z_i = X_i(\text{cross})\) be the iterations that have not. \(\mathbb{E}[\sum_{i=1}^{n} X_i(\text{cross})^2] = O(\mathbb{E}[\sum_{i=1}^{n} Y_i^2]) + O(\mathbb{E}[\sum_{i=1}^{n} Z_i^2])\).

When phase \(j\) starts, let us fix the set of names in \(G_{j' \geq j}\) as \(\Gamma\), and let us also fix set
of at most $\frac{n}{2^{r-1}}$ (due to Lemma 3.2.4) ongoing iterations that have not yet picked a spot at line 28. Claim 3.2.14 gives $E[\sum_{u_i \in G_j} Z_i^2] \leq E[\sum_{u_i \in G_{j'} \geq j} Z_i^2] = O(\frac{n}{2^{r-1}})$, implying $E[\sum_{i=1}^{n} Z_i^2] = O(n)$.

Consider any execution of the algorithm against the adversary. As the adversary strategically schedules the steps of the processors and message deliveries, at each point of execution, we again maintain a configuration of the system $C$. If no more than half of the processors view name $u_i$ as contended, then $C[u_i]$ as before denotes the number of processors that selected $u_i$ when they last picked a spot at line 28. At the moment when name $u_i$ becomes contended in the views of more than half of the processors, we fix $C[u_i]$ and never change it again ($C[u_i]$ still includes all processors that pick $u_i$ exactly at that moment). The potential $H_C$ at any point is again defined based on the actual configuration $C$: $H_C = \sum_{u_i} (\max(C[u_i] - 2, 0))^2$.

Let us argue that $Y_i \leq C[u_i]$. Until the point when more than half of the processors view $u_i$ as contended, $C[u_i]$ counts all iterations where $u_i$ is selected. We just need to show that any iteration that picks $u_i$ at line 28 strictly after the first time point when more than half processors view $u_i$ as contended, cannot by definition be an iteration contributing to $Y_i$. At the time when more than half of the processors first view a name $u_i$ as contended, which group contains $u_i$ is already fixed, let us say $u_i \in G_j$. The iteration that later picks $u_i$ may not start after phase $j$ ends (or else it would view $u_i$ as contended). If it started in phase $j$, then this iteration is clean($j$) or dirty($j$) and does not count toward $X_i$(cross). Otherwise, the iteration started in an earlier phase than $j$, but picked the spot strictly after the time when more than half processors first viewed $u_i$ as contended, i.e. strictly after phase $j$ started. As such, the iteration counts towards $Z_i$, not $Y_i$.

Let $H_f$ be the maximum expected potential the adversary can achieve when execution finishes, where maximum is taken over adversarial strategies. The above argument implies that $E[\sum_{i=1}^{n} Y_i^2] = O(n) + O(H_f)$. We say that a processor $p$ is on a spot $u_i$, if the last time $p$ executed line 28 it picked $u_i$. To upper-bound $H_f$, let us again consider the spot selection events by processors at line 28 in the order they happen one-by-one breaking ties arbitrarily. We update the number of processors on
spots accordingly at each event, even if some events happen simultaneously. At the time of spot selection by some processor \( p \), let target spots be spots that:

- were viewed as uncontended by \( p \)
- are not yet viewed as contended by more than half of the processors (i.e. the time point when this happens first time is strictly after the current spot selection by \( p \)).
- each have either zero or at least two processors on them.

As before, let us call an iteration meaningful if the processor \( p \) picks one of the target spots. Then, both Claim 3.2.11 and Claim 3.2.12 still hold if we set \( j = 1 \), because at the time of the meaningful iteration the processors on some spot \( u_i \) have not finished the iteration in which they picked \( u_i \) (otherwise, they would have propagated contention information and more than half of the processors would view \( u_i \) as contended, contradicting the definition of the target spot).

Now, for \( \alpha > 4 \), the probability that the adversary can execute \( \alpha n \) meaningful iterations is at most \( e^{\frac{\alpha n}{8}} \). Finally, Claim 3.2.13 also holds for the current definitions and analogous to the derivation in Lemma 3.2.10, the maximum expected potential \( H_f \) is at most \( 12n + \sum_{i=4n}^{\infty} e^{\frac{i}{8}} (3l) = O(n) \).

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

**Theorem 3.2.16.** The expected message complexity of our renaming algorithm is \( O(n^2) \).

**Proof.** We know \( X_i = X_i(\text{clean}) + X_i(\text{dirty}) + X_i(\text{cross}) \) and by Muirhead’s inequality \( \mathbb{E}[\sum_i X_i^2] \leq O(\mathbb{E}[\sum_i X_i(\text{clean})^2]) + O(\mathbb{E}[\sum_i X_i(\text{dirty})^2]) + O(\mathbb{E}[\sum_i X_i(\text{cross})^2]) \).

By Lemma 3.2.8, Lemma 3.2.10 and Lemma 3.2.15 we get \( \mathbb{E}[\sum_i X_i^2] = O(n) \). Combining with Lemma 3.2.3 gives the desired result.

The round 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 become assigned during that iteration.
Theorem 3.2.17. The round complexity of the renaming protocol is $O(\log^2 n)$, with high probability.

Proof. In the following, 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 $i$th iteration of the loop, and let $m_i = |M_i|$. By construction, notice that there can be at most $m_i$ processors that compete with for slots in $M_i$ for the rest of the execution.

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 being low-information if $y_i/m_i < 1/\log m_i$. Notice that, in an iteration that is high-information, the processor might collide, but at least reduces its random range for choices by a $1/\log m_i$ factor.

Let us now focus on low-information iterations, and in particular let $i$ be such an iteration. Notice that we can model the interaction between the algorithm and the adversary in iteration $i$ as follows. Processor $p$ first makes a random choice $r$ from $m_i$ slots it sees as available. By the principle of deferred decisions, we can assume that, at this point, the adversary schedules all other $m_i - 1$ processors to make their choices in this round, from slots in $M_i$, with the goal of causing a collision with $p$'s choice. (The adversary has no interest in showing slots outside $M_i$ to processors.) Notice that, in fact, the adversary may choose to schedule certain processors multiple times in order to obtain collisions. However, by construction, each re-scheduled processor announces its choice in the iteration to a quorum, and this choice will become known to $p$ in the next iteration. Therefore, re-scheduled processors should not announce more than $m_i/\log m_i$ distinct slots. Intuitively, the number of re-schedulings for the adversary can be upper bounded by the number of balls falling into the $m_i/\log m_i$ most loaded bins in an $m_i-1$ balls into $m_i$ bins scenario. A simple balls-and-bins argument yields that, in any case, the adversary cannot perform more than $m_i$ re-schedules without having to announce $m_i/\log m_i$ new slots, with high probability in $m_i$.

Recall that the goal of the adversary is to cause a collision with $p$'s random choice $r$. We can reduce this to a balls-into-bins game in which the adversary throws
$m_i - 1$ initial balls and an extra $m_i$ balls (from the re-scheduling) into a set of $m_i(1 - 1/\log m_i)$ bins, with the goal of hitting a specific bin, corresponding to $r$. (The extra $(1 - 1/\log m_i)$ factor comes from the fact that certain processors (or balls) may already observe the slots removed in this iteration.) The probability that a fixed bin gets hit is at most

$$\left(1 - \frac{1}{m_i(1 - 1/\log m_i)}\right)^{2m_i} \leq \left(\frac{1}{e}\right)^3.$$  

Therefore, 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 high probability, either because $m_i = 1$ or because one of its probes was successful in a low-information phase.
Chapter 4

Message Complexity Lower Bounds

Theorem 4.0.18. If \( k \) processors participate in any \( ([n/2] - 1) \)-resilient, linearizable Test-and-Set algorithm, the adversary can guarantee an execution where \( \Omega(kn) \) messages are exchanged.

Proof. We now assume the role of an adversary, and implement the following strategy: we pick an arbitrary subset \( A \) of \( [k/4] \) participants, and place them in a bubble: for each such processor \( q_i \), we suspend all its incoming and outgoing messages in a buffer \( B_i \), so that neither incoming nor outgoing messages reach their recipient, until there are at least \( n/4 \) messages in the buffer \( B_i \). At that point, we free the processor \( q_i \) from the bubble and allow it to take steps synchronously, together with other processors. Processors outside \( A \) execute in lock-step, and we deliver their messages in a timely fashion.

Based on this strategy, we obtain a family of executions \( \mathcal{E} \), which only differ in the coin flips obtained by the processors, and their impact on the processors’ actions. Assume for contradiction that there exists an execution \( E \in \mathcal{E} \) in which exists a processor \( q_j \in A \) which is still in the bubble at the end of the execution. Practically, in every synchronous execution in which all messages to and from the set \( A \) get arbitrarily delayed until the processor sends or receives more than \( n/4 \) messages, process \( q_j \) sends at most \( n/4 \) messages, and receives none.

Claim 4.0.19. Processor \( q_j \) cannot return in execution \( E \).
To prove the claim, notice that we can build two alternate executions $E'$ and $E''$ which are indistinguishable from $E$ to $q_j$, with the following structure. First, all processors flip the same coins in $E', E''$ as in $E$. In $E'$, the adversary executes only steps by $q_j$, and suspends all other processors, crashing all the recipients of messages from $q_j$. Notice that this does not exceed the failure bound $\lceil n/2 \rceil - 1$. Also, by the semantics of test-and-set, $q_j$ must necessarily return $WIN$ in $E'$. In $E''$, the adversary synchronously executes all processors other than $q_j$ until they return, crashing all processors which send messages to $q_j$. Since this does not exceed the failure bound $\lceil n/2 \rceil - 1$, by the termination property of the implementation, there exists a finite time after which all processors return. The adversary then runs processor $q_j$ until completion. Notice that, by the specification of test-and-set, processor $q_j$ must return $LOSE$ in $E''$.

Finally, notice that executions $E, E', E''$ are indistinguishable for processor $q_j$, since it receives no messages in any of them, and flips the same coins. The contradiction follows since $q_j$ must then decide two distinct values based on indistinguishable states.

We have therefore proved that, if a processor $q_j \in A$ does not receive any messages during the execution due to our buffering strategy, then, with probability 1, the processor cannot return. To complete the argument, we prove that this is not a valid outcome for $q_i$, given the task specification. Note that the execution $E$ may not be a valid execution in this model, since not all messages by correct processors have been delivered. (In particular, $q_i$’s messages have not been yet delivered.)

**Claim 4.0.20.** There exists an valid execution $F$ in this model, which is indistinguishable from $E$ for $q_i$.

We build the execution $F$ as follows. Let $\tau$ be time in $E$ after which no more messages are sent. (If no such time exists, then our lower bound trivially follows.) After this time, we deliver all of $q_i$’s messages, and crash the recipients immediately thereafter. Also, we crash all processors which sent messages to $q_i$, and do not deliver their messages. Clearly, $F$ contains less than $n/2$ faults, and is indistinguishable to
$q_i$ from $E$. We therefore obtain that $q_i$ does not return in any execution $F$ built from $E \in \mathcal{E}$, a contradiction. We have therefore obtained that any processor $q_j \in A$ must send or receive at least $n/4$ messages in every execution under our strategy, therefore the expected message complexity of the algorithm is $\Omega(kn)$, as claimed. \[ \square \]

**Theorem 4.0.21.** In any $(\lceil n/2 \rceil - 1)$-resilient Renaming algorithm, the adversary can guarantee an execution where $\Omega(n^2)$ messages are exchanged.

**Proof.** The proof is completely analogous to the proof of Theorem 4.0.18 with $k = n$, and the following minor modification: we have to prove that in a renaming algorithm, in a valid execution, a non-faulty processor cannot return without receiving any messages. Consider the set of possible initial names, which come from an unbounded domain. For some of these names, processors starting with them could in principle return without receiving any messages. However, there can not be more than $n$ such names: otherwise, there would inevitably be two original names, such that processors with these names could return the same name in finite time without receiving any messages. Then, an execution would be possible, whereby the first processor executes and returns, then second runs and returns while all messages are being delayed, and they both get the same name, contradicting the correctness of the algorithm. Therefore, the adversary simply has to choose participants in the protocol with initial names that are different from these at most $n$ special name. Then, as shown above, a processor that does not receive a message, can not possibly return, completing the proof. \[ \square \]
Chapter 5

Discussion and Future Work

We have given tight bounds for the message complexity of leader election (test-and-set) and renaming in an asynchronous message-passing model, against an adaptive adversary. Our test-and-set algorithm terminates in \(O(\log \log n)\) steps (communication rounds), which is exponentially less than the step complexity of the fastest known shared-memory algorithm in this model. This is reasonable since the ability to send messages in parallel naturally reduces time complexity. It also has interesting implications for lower bounds on the time complexity of randomized test-and-set, which remains an intriguing open problem [AAG+10, GW12a].

In particular, it suggests that classic topological techniques, e.g. [HS99], cannot be used to obtain an \(\Omega(\log n)\) lower bound on the time complexity of test-and-set with a strong adversary, since these tools assume that snapshots have unit cost—our algorithm can be easily translated to this model, maintaining its round (step) complexity. Our algorithm suggests that a logarithmic lower bound would have to exploit the cost of disseminating information in shared-memory. Determining the tight time complexity bounds for these objects remains an intriguing open question.

Another interesting research direction would be to apply the tools we developed to obtain message-efficient implementations of other fundamental distributed tasks, such as task allocation or mutual exclusion, and to explore solutions with lower bit complexity. In particular, we have some preliminary results for the At-Most-Once problem [KKNS09], which is an important setting of task allocation.
Appendix A

Test-And-Set for Two Processors

A.1 MiniTAS algorithm

Our two-processor test-and-set algorithm, called MiniTAS is, just like the others, \(\lceil n/2 \rceil - 1\)-resilient and linearizable. It has message complexity \(O(n)\) and round complexity \(O(1)\), in expectation.

We note that a classical implementation of this algorithm for shared memory is the one due to Tromp and Vitányi [TV02], who did an extensive analysis in order to prove correctness. Our algorithm can, essentially, be considered a simplification from the aforementioned; furthermore our exposition offers a slightly more unified approach, since we use the same ideas of first announcing participation (which turns out to be crucial), then broadcasting a priority in a different round of communication, just like in the PoisonPill algorithm.

For presentation purposes, we can assume that the two processors know each other’s id’s. This can be trivially extended to the general case, where at most two processors ever compete in MiniTAS, but without knowing the competitor’s id in advance.

At every iteration through the loop, the participating processors set their state to \textit{Commit}, then propagate it to a quorum. After having announced participation this way, a processor flips a coin in order to set its status to \textit{High-Pri} or \textit{Low-Pri}. Then it propagates its status again, and finally collects information from the quorum.
about the status of the competitor. Processors append each status with a strictly increasing stamp value before propagating. When a processor \( p \) collects views of the status of its competitor \( q \) from a quorum, it selects the unique status with the highest stamp (which will be the at least as recent as the last status that \( q \) propagated to a quorum). If the status of a processor \( p \) is different from this status of the competitor \( q \), \( p \) will break out of the loop at line 37 and return WIN or LOSE based on its priority. Otherwise, it will repeat the iteration.

Note that the moment a processor has returned WIN or LOSE, it continues sending and receiving messages by propagating its final priority. However, it will never change its status again.

The MiniTAS algorithm is described in Figure A-1, and following section contains more formal description and analysis.

**Input:** Unique identifier \( i \) of the participating processor, \( j \) is the id of the competitor

**Output:** WIN or LOSE (result of the Test-and-Set)

**Local variables:**
- \( Status[n] = \{\{Low-Pri, 0\}\} \)
- \( Views[n] = \{\bot\} \)

```plaintext
procedure MiniTAS(i)
    stamp ← 1
    competitorStatus ← Low-Pri
    while (Status[i] = competitorStatus) do
        Status[i] ← \{Commit, stamp\}
        stamp ← stamp + 1
        communicate(propagate, Status[i])
        Status[i] ← \{random(Low-Pri, High-Pri), stamp\}
        stamp ← stamp + 1
        communicate(propagate, Status[i])
        Views ← communicate(collect, Status[j]) /* Processor considers only the status with the highest stamp */
        competitorStatus ← (Views[k].first | \( \forall k' : Views[k'].second \geq Views[k]\).second)
        if Status[i] = High-Pri then return WIN
    else return LOSE
```

Figure A-1: Pseudocode of the MiniTAS algorithm for two processors.
A.2 Analysis

**Theorem A.2.1.** MiniTAS is a linearizable Test-and-Set algorithm, resilient to \( \lceil n/2 \rceil - 1 \) process faults. The message complexity is \( O(n) \) and the round complexity is \( O(1) \), in expectation.

**Lemma A.2.2.** If both participating processors return in MiniTAS, one of them will return WIN and the other will return LOSE.

*Proof.* First we remark that once a processor returns, it will never change its status again. It will be set to *High-Pri* or *Low-Pri*, and the processor will continue communicating with the other processors, but this will not affect its status any more.

Let us equivalently prove that both processors cannot break out from the loop at line 37 with the same priority. The proof is by contradiction. Consider the last time one of the processors, say \( p \) started to execute line 44. At that point, the other processor \( q \) has already started executing 44, meaning that \( q \) has already finished propagating its final priority (final because by our assumption this is \( q \)'s last while loop iteration). Therefore, \( p \) will pick \( q \)'s final priority as its *competitorStatus* and in order to break out of the loop itself, its own status, which is the same as \( p \)'s final priority, has to be different from \( q \)'s final priority.

**Lemma A.2.3.** MiniTAS is a linearizable Test-and-Set algorithm.

*Proof.* Let us first show that a processor can always be linearized to return WIN in MiniTAS. If both processors participate, this follows from Lemma A.2.2. Otherwise, observe that if only one processor runs in isolation it can only ever break out of the while loop with high priority, thus it can only return WIN.

Now consider the winning processor \( p \), and linearize its execution to its invocation point \( P \). Let \( q \) be the other participating processor (if it exists). Then we can linearize its execution after \( P \). Suppose this was not possible. Then the execution interval of \( q \) must start and finish before the invocation point \( P \). But this implies that \( q \) returns WIN. This contradicts our linearization and Lemma A.2.2. 

\( \square \)
Lemma A.2.4. Every processor that participates in MiniTAS returns after \( O(1) \) rounds of communication, in expectation.

Proof. Since every processor will communicate three times in every iteration of the while loop, we only need to bound the number of iterations.

If there is only one processor running alone then it will win, in expectation, after 2 iterations. Indeed, at every iteration it picks a random priority, then it collects information. Line 37 will evaluate to false with probability \( \frac{1}{2} \), in which case the loop will finish and the processor will return WIN. Otherwise, it will simply go to the next iteration and repeat.

If there are two participating processors, \( p \) and \( q \), then we only need to argue that one of them will return after \( O(1) \) iterations. Suppose that \( p \) returns first. For every new iteration (unless \( q \) returns in the same iteration it was in when \( p \) did its final collect) when \( q \) collects statuses, it will see \( p \)'s priority. Since \( q \) chooses a priority opposite to that of \( p \) with probability \( \frac{1}{2} \), it returns with probability \( \frac{1}{2} \). Therefore, \( q \) returns after \( O(1) \) extra iterations.

A processor will keep iterating for as long as the expression at line 37 will evaluate to true. Let us understand why the adversary can not force this to happen for too long. In order for the processors to keep iterating, it must be the case that whenever either \( p \) or \( q \) passes through line 37, it observes that competitorStatus is the same as its own status.

At any time, for processor \( q \) there exists status \( s_1 \) with the highest stamp, that has been propagated to a quorum, and potentially another status \( s_2 \) with one higher stamp, that has been propagated to some, but not all processors. Because every status with the odd stamp is Commit, one of the statuses \( s_1 \) and \( s_2 \) is a Commit status.

Consider the time when processor \( p \) flips a coin to choose its new priority at line 41. With probability at least \( \frac{1}{2} \), it will be different from both \( s_1 \) and \( s_2 \) of processor \( q \) at the moment it is chosen. In this case, if \( q \) does not propagate any more statuses until \( p \) finishes collecting at line 44, then \( p \) will set competitorStatus to either \( s_1 \) or \( s_2 \). Both \( s_1 \) and \( s_2 \) are different from \( p \)'s status, so \( p \) will finish iterating through the
loop and return.

Therefore, the only way to force \( p \) to keep iterating is to make \( q \) change its priority. This means that before \( p \) advances to the next iteration, \( q \) must iterate until it picks a priority that is different from both \( s_1 \) and \( s_2 \). However, in each iteration, \( q \) picks a high or low priority once, before evaluating line 37 itself. With probability 1/2, it will pick the same priority as it had before, which will not match the priority status of \( p \) (\( p \) had already propagated Commit status to a quorum, and potentially an opposite priority information with one larger stamp, both of which do not match). In this case \( q \) will return. Hence with probability at least 1/4, both \( p \) and \( q \) will both return.

At every iteration, one of the processors returns with probability at least 1/4. So in expectation there will be only a constant number of rounds before one of the processors returns (followed by the other processor, if it participates).

Together with the fact that every round, a processor sends only \( O(n) \) messages, we conclude that during the execution of MiniTAS, \( O(n) \) messages are exchanged, in expectation. \( \square \)

**Lemma A.2.5.** MiniTAS is resilient to \( \lceil n/2 \rceil - 1 \) processor faults.

**Proof.** This is a standard argument and it follows from the fact that processors communicate through a quorum. \( \square \)
Appendix B

Adaptive Test-and-Set

B.1 AdaptiveTAS Algorithm

The adaptive Test-and-Set algorithm relies on the MiniTAS algorithm presented in Appendix A. Our approach is based on extending the idea of Fischer and Peterson [PF77] of building a tournament tree, where the nodes correspond to running test-and-set between pairs of processors. The RatRace protocol from [AAG+10] made this algorithm adaptive, i.e. the message and round complexity depend only on the number of competitors, not on the total number of processors. In order to achieve the claimed guarantees, we also require adaptiveness.

Just like in [AAG+10] the tournament tree is built using the randomized splitter object, as defined in [AKP+06]. Let us recall how a splitter works. A processor that enters the splitter returns stop, left, or right. If only one processor enters the splitter, it is guaranteed to stop. If there are two or more processors entering the splitter, at most one of them returns stop, while the rest return a value of left or right, independently and uniformly at random.

The construction of the tree goes, roughly, as follows. Initially all the participating processors enter the splitter corresponding to the root of the tree. If a processor returns stop, then it acquires the splitter, and halts at the current node. The others are randomly assigned to left or right. The assigned value tells every processor on which side of the tree they should descend at the next step. The processors continue
As soon as a processor has acquired a splitter, it starts competing against other processors in order to climb back up the tree. Namely, the processors that have won in the left, respectively right subtrees rooted at the children of the current node play against each other in MiniTAS. The winner plays against the processor that may be at the current node. The winner among these at most three processors acquires the current node, and continues up the tree recursively. The processor who wins at the root of the tree at the end is declared winner.
For completeness, we sketch the algorithm in Figure B-1. We used \( \text{prefix}_x(y) \) and \( \text{suffix}_x(y) \) to denote the length \( x \) prefix (respectively suffix) of string \( y \). Note that, in addition to the standard specifications from MiniTAS, here we call the test-and-set object using an additional label in the subscript. The purpose of it is to specify who participates in a given test-and-set, although messages corresponding to different test-and-set objects are being passed simultaneously.

### B.2 Analysis

**Lemma B.2.1.** The expected number of nodes in the tournament tree is at most \( 2^k \).

**Proof.** Let \( f(n) \) denote the expected number of nodes in a tree corresponding to \( n \) processors. As base steps, we have \( f(0) = 0 \) and \( f(1) = 1 \). The latter holds because when there is only one processor, it halts at the splitter corresponding to the root. Otherwise, the current node may be left unoccupied, and the \( n \) processors get distributed independently into the two subtrees rooted at the current node. Let \( b(j; k) \) be the probability that out of \( k \) processors, \( j \) of them end up in the left subtree (and recall that \( b(j; k) = 2^{-k \binom{k}{j}} \)).

Conditioning the expectation \( f(k) \) on the number of nodes that get distributed into the left subtree we get the recurrence:

\[
\begin{align*}
  f(k) &\leq 1 + b(k; k)(f(k) + f(0)) + b(k - 1; k)(f(k - 1) + f(1)) \\
 &\quad + \cdots + b(1; k)(f(1) + f(k - 1)) + b(0; k)(f(0) + f(k)) \\
 &\quad = 1 + 2 \sum_{j=0}^{k} b(j; k)f(j)
\end{align*}
\]

Which is equivalent to

\[
 f(k) \leq \frac{1 + 2 \sum_{j=0}^{k-1} b(j; k)f(j)}{1 - 2^{-k+1}}
\]

It can be easily shown by induction that \( f(k) \leq 2k - 1 \). Assume this holds up to
Lemma B.2.2. The expected height of the tournament tree is $O(\log k)$.

Proof. Just like in Lemma [B.2.1] let $g(k)$ be the expected height of a tree corresponding to $k$ processors. Clearly, $g(0) = 0$ and $g(1) = 1$. The recurrence is similar, except for the fact that instead of using the fact that the number of nodes in a subtree equals 1, plus the sum of the number of nodes in the subtrees induced by the root’s children, we have that the height of a subtree is equal to 1 plus the maximum of the heights of the subtrees induced by the root’s children. The expression below should be self-explanatory.

\[
g(k) \leq 1 + b(k; k) \max(g(k), g(0)) + b(k - 1; k) \max(g(k - 1), f(1)) + \cdots + b(1; k) \max(g(1), g(k - 1)) + b(0; k) \max(g(0), g(k))
\]

We can upper bound $g(k)$ by breaking the expectations into the event where the processors at the current vertex get split into two balanced parts, with sizes within $[k/4, 3k/4]$, and the event where one of the branches of the tree will still contain more
than $3k/4$ processors.

$$g(k) \leq 1 + \sum_{j \in [k/4,3k/4]} b(j;k)g(3k/4) + \sum_{j \not\in [k/4,3k/4]} b(j;k)g(k)$$

$$\leq 1 + g(3k/4) + e^{-\Theta(k)} \cdot g(k)$$

The last inequality follows from applying a Chernoff bound on the tail of a sum of Bernoulli random variables. Solving the recurrence yields the desired result $g(k) \leq O(\log k)$.

**Theorem B.2.3.** AdaptiveTAS is a linearizable Test-and-Set algorithm, resilient to $\lceil n/2 \rceil - 1$ process faults. If $k$ processors execute AdaptiveTAS, then the message complexity is $O(kn \log k)^1$ and the round complexity is $O(\log k)$.

**Proof.** First we bound the expected number of exchanged messages, and the expected number of communicate calls made by any processor.

We bound separately the messages communicated by processors in each of the two phases of the algorithm: the phase when processors are propagated down the tree through splitters, and the phase where processors climb their way back up towards the root.

In the first phase, each processor passes through a number of splitters, until it acquires a leaf of the tree. By Lemma B.2.2, each processor will pass through $O(\log k)$ splitters, in expectation. Upon entering a splitter, it communicates $O(n)$ messages with the other processors in $O(1)$ rounds of communication. Hence in the first phase, each participating processor will communicate $O(n \log k)$ messages in $O(\log k)$ rounds.

For the second phase, notice that every node of the tree corresponds to a constant number of test-and-set routines, each of them requiring $O(n)$ messages. Also, the algorithm requires at most two calls to MiniTAS at every node of the tree, and from Theorem A.2.1 we know that each of these calls sends at most $O(n)$ messages. Also,

---

1We suspect that the number of messages can be reduced to $O(kn \log \log k)$ by introducing some additional tricks, but we will not worry about this here, since $O(kn \log k)$ messages are sufficient for our purposes.
in Lemma B.2.1 we have shown that the number of nodes in the tree is $O(k)$. Hence the total number of sent messages is at most $O(kn)$, in expectation.

Now observe that every processor plays MiniTAS only against the processors it encounters on its way up towards the root, and it runs the routine at most twice at every node. From Lemma B.2.2 we know that this will happen at most $O(\log k)$ times for every processor. From Theorem B.2.3, we know that every call of the routine requires $O(1)$ calls to the communicate procedure.

Putting everything together, we get that the round complexity of AdaptiveTAS is $O(\log k)$, while the message complexity is $O(kn \log k)$.

The AdaptiveTAS algorithm can be made linearizable by implementing a doorway mechanism, similar to the one used in PoisonPill. Linearization will then quickly follow by emulating the proof from [AAG+10]. \qed
Bibliography


