UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Fault-Tolerant Renaming in Synchronous Message-Passing Systems

Oksana Denysyuk

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD degree in Information Systems and Computer Engineering
Jury final classification: Pass with Distinction

2014

UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Fault-Tolerant Renaming in Synchronous Message-Passing Systems

Oksana Denysyuk

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

Funding Institutions

Fundação para a Ciência e a Tecnologia

2014

Resumo

Os problemas de coordenação são centrais no desenvolvimento de sistemas distribuídos. Esta tese aborda o problema de coordenação designado por *renomeação*. Neste problema, um conjunto de processos, com identificadores pertencentes a um vasto espaço de nomes, escolhe novos identificadores confinados a um espaço de nomes muito mais reduzido. O problema de renomeação pode ser visto como complementar do clássico problema do consenso distribuído: em vez de *concordar* num único valor, na renomeação todos os processos têm de *discordar* de forma construtiva, escolhendo identificadores distintos dentre um conjunto restrito de valores alvo.

A tese foca-se em sistemas distribuídos síncronos, onde n processos estão interligados por uma rede totalmente conexa e comunicam exclusivamente através da troca de mensagens. Neste contexto, estudam-se soluções para a renomeação na presença de falhas por paragem e na presença de falhas arbitrárias.

Denote-se por t o número máximo de processos que pode falhar. Considerando falhas por paragem, a tese mostra que, recorrendo a fontes de aleatoridade, a renomeação pode ser resolvida em $O(\log \log n)$ rondas com alta probabilidade em sistemas onde $t < n$. Em contraste, os algoritmos deterministas necessitam de $\Omega(\log n)$ rondas [CHT99]. Portanto, o nosso resultado implica que os algoritmos probabilísticos de renomeação podem superar algoritmos deterministas por um factor exponencial em n. Considerando falhas arbitrárias, a tese mostra que os algoritmos probabilísticos podem resolver renomeação em $O(\log n)$ rondas com alta probabilidade para $t < n$ (em contraste, os algoritmos deterministas necessitam de um número ilimitado de rondas [OBG08]). Estes resultados implicam que o recurso à aleatoriedade é uma técnica poderosa para resolver o problema da renomeação.

Ainda no contexto de falhas arbitrárias, a tese aborda pela primeira vez o problema da renomeação com *preservação de ordem*. Nesta variante do problema de renomeação, os processos recebem novos nomes que respeitam a ordem dos seus identificadores originais. A tese mostra que renomeação com preservação de ordem pode ser resolvida em $O(\log n)$ rondas, para $t < n/3$. Adicionalmente, a tese mostra que renomeação com preservação de ordem pode ser resolvida em $O(1)$ rondas quando $n > t^2 + 2t$. Finalmente, a tese demonstra que este problema é impossível de resolver para $t \geq n/3$; a impossiblidade aplica-se tanto a algoritmos deterministas, como a algoritmos com recurso a aleatoriedade. Este resultado de impossibilidade implica uma separação entre o problema de renomeação original e o problema com preservação de ordem no modelo de falhas arbitrárias.

Abstract

Exploring the power and limitations of different coordination problems has always been at the heart of the theory of distributed computing. This thesis addresses the coordination problem called *renaming*, in which processes start with ids from a large namespace and output new names from a small namespace. Renaming can be seen as a dual to the classical consensus problem: instead of *agreeing* on a unique value, in renaming correct processes must *disagree* in a constructive way, by picking distinct values (names) from an appropriate range of values.

We focus on synchronous message-passing systems, where n processes are arranged in a fully connected network and communicate by sending messages. We show that, with resort to randomization, renaming can be solved in $O(\log \log n)$ rounds with high probability, if up to $t < n$ may fail by crashing (t is an upper bound on the number of faults). In contrast, deterministic algorithms require $\Omega(\log n)$ rounds [CHT99]. Thus, our result implies that randomized renaming algorithms can outperform deterministic algorithms by an exponential factor. When considering Byzantine faults, we show that randomized algorithms can solve renaming in $O(\log n)$ rounds with high probability, while tolerating up to $t < n$ Byzantine faults (in contrast, deterministic algorithms require unbounded time [OBG08]). These results imply that randomization is a powerful and necessary technique in solving renaming.

Finally, this thesis is the first to address *order-preserving* renaming in the context of Byzantine faults. In order-preserving renaming, processes must output new names in the order of their original ids. We show that renaming can be solved in $O(\log n)$ rounds, if up to $t < n/3$ processes may be Byzantine. We also show that order-preserving renaming can be solved in $O(1)$ rounds if the maximum number of Byzantine faults is bounded by the formula $n > t^2 + 2t$. On the negative side, we show that order-preserving renaming cannot be solved if $t \geq n/3$; the impossibility applies to both deterministic and randomized algorithms. This result establishes a separation between the resiliency of renaming and order-preserving renaming algorithms in the Byzantine fault model.

Palavras-Chave

Keywords

Palavras-Chave

Renomeação

Renomeação com preservação de ordem

Tolerância a falhas

Algoritmos probabilísticos

Algoritmos distribuídos

Sistemas síncronos de transmissão de mensagens

Problemas de coordenação

Falhas por paragem

Falhas arbitrárias

Teoria da computação distribuída

Keywords

Renaming

Order-preserving renaming

Fault-tolerance

Randomized algorithms

Distributed algorithms

Synchronous message-passing systems

Coordination problems

Crash faults

Byzantine faults

Theory of distributed computing

Acknowledgments

The author was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the individual Doctoral grant SFRH/BD/69653/2010, via project PEPITA (PTDC/EEI-SCR/2776/2012), and via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/ 2013.

Contents

List of Figures

List of Tables

List of Abbreviations and Notation

AA	Approximate agreement
PKI	Public key infrastructure
whp	with high probability

n	The number of processes in the system
p_i	A process
t	The upper bound on the number of possible faults in the system
f	The number of actual faults in an execution
N_{max}	The largest id among the ids of all processes in the system
M	The size of the target namespace

Chapter 1

Introduction

An essential requirement in distributed computing is the ability to uniquely distinguish processes. Unfortunately, unique identifiers often come from a large namespace (e.g., IP or MAC addresses, email addresses, hash values), which can be undesirable. For example, in some distributed algorithms, the running time is sensitive to the numerical value of the process ids, and large ids can significantly slow down the computation [Dij82]. In other algorithms, unique ids are embedded in messages, and large ids can increase the message size. These considerations motivated the *renaming* problem, which has become a fundamental problem in distributed computing.

1.1 Renaming

In renaming, processes start with unique identifiers from a large namespace and output new names from a smaller namespace. Intuitively, renaming can be seen as a dual to the classical *consensus* problem: on one hand, solving consensus requires all correct processes *agree* on a single value; on the other hand, renaming requires correct processes *disagree* in constructive way, by picking distinct values (names) from a small range of values.

More precisely, in renaming n processes start with distinct identifiers taken from an unbounded *original namespace*, and output distinct names from a smaller *target names-*

pace of size M, where M is the parameter of the problem determined *a priori*. If the size of the target namespace is exactly n (which is the smallest possible), then the problem is known as *tight* (or *strong/perfect*) renaming. When solving renaming, it is sensible to aim at the smallest target namespace; however, as we will discuss in the related work, tight renaming is not always possible to achieve in certain systems. Therefore, in addition to the standard concerns about the complexity (the amount of resources it takes to solve the problem) and resiliency (what assumptions on the number and type of faults we must make), the size of the new namespace is also an important dimension to the renaming problem.

In addition to renaming, this thesis addresses the *order-preserving* variant of the problem [ABND$^+$90], which requires new names preserve the order of the original ids (i.e., if the id of process p_i is smaller than the id of process p_j, then the new name of p_i must be smaller than the new name of p_j). This variant is interesting in settings where the original identifiers encode some additional information, such as their relative priority in accessing a shared resource.

Though originally renaming was introduced for the message-passing model, the majority of research effort has been devoted to the study of this problem in the shared memory model, e.g. [BG93, AF02, CR10, CnR12, AACH$^+$14]. As a result, many fundamental questions on solving renaming in the message-passing model remained unanswered.

In this thesis we focus our attention on the synchronous message-passing model and aim at contributing to a better understanding of renaming and order-preserving renaming problems in this model. We put the hypothesis that renaming and order-preserving renaming can be solved both efficiently and with high resiliency to different types of failures. Below we summarize our findings that support this hypothesis. The material presented in this thesis is based on the publications listed at the end of the document.

1.2 Summary of Contributions

In this section, we describe the main contributions of the thesis. Please see Tables 1.1 and 1.2 at the end of this chapter for a schematic summary of these results.

1.2.1 Renaming with Crash Faults

In Chapter 4, we assume that processes are prone to crashes and up to $t < n$ processes may crash. Previous work by Chaudhuri, Herlihy, and Tuttle [CHT99] gave an elegant algorithm that solves *tight* renaming in this setting with $O(\log n)$ round complexity. The authors also showed this to be optimal by proving an $\Omega(\log n)$ lower bound for algorithms which distinguish state through comparisons (see Chapter 2). This result concerns only deterministic algorithms and leaves open the possibility of achieving better, sub-logarithmic solutions by resorting to randomization.

Randomization is a natural approach for renaming, and has been used to achieve low-complexity solutions in the related shared-memory model, e.g. [AACH$^+$14]. Moreover, since renaming can be seen as an assignment problem, in which n resources (names) must be uniquely assigned to n processes, randomized renaming is related to the extensive line of work on randomized load balancing, e.g. [ACMR95, BKSS13, LW11b, Mit01], for which different sub-logarithmic solutions have been proposed. Surprisingly, a careful analysis of such sub-logarithmic solutions reveals that none of them can be used to achieve tight renaming: they either are designed for a fault-free setting, or relax the one-to-one allocation requirement. It is therefore an intriguing question whether randomization can in fact be used to solve fault-tolerant tight renaming in sub-logarithmic time.

We answer this question in the affirmative. We present a new algorithm to solve tight renaming in $O(\log \log n)$ communication rounds with high probability, exponentially faster than the optimal deterministic counterpart. The algorithm, which we call *Balls-into-Leaves*, is based on the idea of arranging the target names as leaves of a binary tree; processes start at the root of the tree, and perform repeated random choices in order to disperse themselves towards leaves, while minimizing contention. We also present an early-terminating variant, which terminates in $O(\log \log f)$ rounds, where f is the actual number of failures in the execution, which is again exponentially faster than the best known deterministic version [AAGT12].

More precisely, our algorithm works as follows: processes (balls) start at the root of a virtual binary tree, whose n leaves correspond to the target names. In each iteration, each ball picks a random available leaf and broadcasts its choice; in case of collisions, a deterministic rule is used to select a winner. The remaining balls backtrack towards the root, stopping at the lowest node at which the ball can still find an available leaf within the corresponding subtree. (Please see Figure 4.1 for an illustration.) Balls then iterate this procedure, exchanging information and progressively descending in the tree.

Our main technical contribution is to show that this natural strategy is fault-tolerant, and in fact converges extremely quickly to a perfectly balanced allocation. Our proof argument has two parts: we first exploit the concentration properties of the binomial distribution to bound the maximum contention on a leaf after $\Theta(\log \log n)$ communication rounds to be $O(\text{polylog } n)$, with high probability (whp). Second, we fix a root-to-leaf path, and prove via a technical argument that all $O(\text{polylog } n)$ balls, except at most one, will disperse themselves off the path within the next $O(\log \log n)$ rounds, again whp. Therefore, each ball reaches a leaf within $O(\log \log n)$ rounds whp.

Since the number of leaves matches the number of balls, Balls-into-Leaves solves *tight renaming*. Given that comparison based deterministic algorithms take $\Omega(\log n)$ rounds [CHT99], this result establishes an exponential separation between such algorithms and their randomized counterparts. (Our algorithm is also comparison-based.) Moreover, Balls-into-Leaves guarantees deterministic termination; in particular, it will complete in a linear number of rounds, even in unlucky runs.

We then extend the Balls-into-Leaves algorithm to ensure *early termination*. Roughly, an early-terminating algorithm terminates faster when there are fewer failures, as its running time is a function of the number f of failures rather than n. We do so by introducing an initial phase that ensures that balls take advantage of a small number of failures in this round, descending deeply into the tree. With this modification, the algorithm terminates in $O(\log\log f)$ rounds whp, where f is the actual number of crashes. Furthermore, in a fault-free execution, it terminates in constant time.

An examination of the proof argument (in particular, the concentration properties of the binomial distribution) suggests that our complexity upper bound for Balls-into-Leaves is in fact tight. The main open question is therefore whether the Balls-into-Leaves strategy is optimal for this problem. We conjecture that answering this question will require a new lower bound technique for randomized renaming.

1.2.2 Renaming with Byzantine Faults

We then consider a more challenging Byzantine fault model. We show that, with the help of randomization, renaming can be solved efficiently for any $t < n$ (Chapter 5). Therefore, our result further confirms that randomization is an important and necessary technique for solving renaming efficiently. To our knowledge, our results are the first to address randomized solutions for renaming in this model.

When considering randomized algorithms in the context of Byzantine faults, the literature distinguishes two adversaries with different powers. With the *non-rushing* adversary, processes are forced to take steps simultaneously at the beginning of each round. Therefore, Byzantine processes must choose messages to send in each round *before* learning the random choices made by the correct processes in the same round. With the *rushing* adversary, Byzantine processes may execute each step after learning about the random choices of the correct processes. In [GY89], non-rushing and rushing adversaries are called simultaneous and sequential, respectively. We study renaming under both adversaries.

We first note that it is possible to solve renaming using a consensus algorithm to agree on a set of identifiers. It is also known that sometimes randomization can be used to solve consensus [BO83] when deterministic algorithms fail [FLP85]. Thus, one might wonder whether we could obtain an efficient randomized renaming algorithm that tolerates $t < n$ Byzantine failures, as follows: first, obtain an efficient randomized consensus algorithm that tolerates $t < n$ Byzantine failures; then use this algorithm to solve renaming. Unfortunately, this approach would not work: it has been shown that with Byzantine failures and $t \geq n/3$, any randomized consensus algorithm fails with probability at least 1/3 [GY89, KY86]. This result holds even with the non-rushing adversary.

We first consider the weaker non-rushing adversary. We propose an algorithm based on a simple idea: each process randomly chooses a new id from the range $\{1, \ldots, n\}$ and applies tie-breaking rules to solve collisions. We show that this algorithm terminates in $O(\log n)$ rounds whp.

We next consider the more challenging *rushing* adversary. We show that our first algorithm can tolerate up to $t = 1$ Byzantine failure but the algorithm fails if $t > 1$. This is because, in each round, Byzantine processes can first observe the choice of a correct process and mimic it, causing infinitely many collisions. To tackle this problem, we use a cryptographic commitment primitive to force processes to commit to a choice without revealing their value; this technique prevents Byzantine processes from constantly mimicking the choices of correct processes. Using cryptographic commitment, we show how to extend the previous algorithm to work for any $t < n$.

To use cryptographic commitment, we must assume a polynomially bounded adversary, so that the adversary cannot break the cryptographic primitive. We believe our use of cryptography is sensible in the following two ways. First, we do not assume a primitive that needs a public key infrastructure (PKI). By contrast, primitives such as *digital signatures*, which are frequently used to cope with Byzantine processes, require a PKI. The problem is that a PKI implies that processes have the public key of all processes, but the public key serves as a unique process id; and if processes have knowledge of unique ids for each other, the renaming problem becomes trivial (e.g., each process sorts these ids, and then outputs as its new name the position of its id in this sorted order).

Second, even if the adversary breaks the cryptographic primitive by luck (by guessing the random seed used by a correct process to commit a value), the algorithm never violates the properties of the renaming problem, though termination may get delayed. Even in this case, termination is ensured with probability one. By contrast, certain algorithms that rely on cryptography will fail with non-zero probability when the adversary is lucky [DS83].

1.2.3 Order-Preserving Renaming with Byzantine Faults

In Chapter 6, we turn our attention to the order-preserving variant of the renaming problem.

In the synchronous model, order-preserving renaming has been previously addressed only in the context of crash faults [Oku10]. In this algorithm, processes help each other to pick new names by running in parallel approximate agreements (AA) on the new names. To obtain a Byzantine tolerant algorithm, one can be tempted to resort to the classical techniques of translating a crash-tolerant algorithm into a Byzantine-tolerant algorithm [BN01, NT88]. However, adapting previous work to cope with Byzantine processes raises several challenges. The first challenge is that original identifiers are not globally known among the processes *a priori*. Note that with this knowledge it becomes trivial to solve order-preserving renaming without any communication (just by sorting the set of ids and then choosing the rank of each id as new name). However, the usual translation techniques assume that the identifiers are known *a priori* to each process; and thus, these techniques cannot be directly applied in our setting. We could further explore the possibility of adapting those translation techniques to cope with the unknown ids and then apply them to the crash-tolerant algorithm of Okun [Oku10]. In fact, Okun, Barak and Gafni took this approach in designing the first Byzantine-tolerant (non order-preserving) renaming algorithm [OBG08]. The downside of this approach is that the translation techniques increase the round complexity by the factor of 4 and introduce a linear increase in the message complexity because processes must broadcast and echo histories of previously received messages.

In contrast, we present a Byzantine-tolerant algorithm that follows the structure of the crash-tolerant algorithm of Okun [Oku10], but with some additional adaptations that do *not* yield any increase in the message and round complexity. Our algorithm consists of two phases:

The purpose of the first phase is to exchange the information about the ids existing in the system. This is achieved by a 3-round id selection scheme that restricts the number of ids in the system, despite lies by Byzantine processes. Our id selection is conceptually similar to the Gradecast [FM88] primitive which assigns confidence levels to messages delivered from other processes.

In the second phase, each process proposes a new name for each known id based on the rank of this id in the ordered set of all identifiers known to this process. The proposed names are then used as inputs into separate instances of Byzantine-tolerant approximate agreement, one per id. Finally, each process returns as its new name the output of the corresponding instance of AA. Byzantine-tolerant AA [DLP+86] guarantees that the outputs are within the range of values issued by the correct processes. However, even after the first phase, Byzantine processes can cause correct processes to propose overlapping intervals of values for different instances of AA. Therefore, without additional measures, the outputs of AA may not preserve the initial ordering. We solve this by simultaneously validating messages from the same process into all concurrent instances of AA. Interestingly, this validation does not require any additional messages.

After presenting the algorithm, we analyze its behavior when n is large compared to t. In the lines of the work for crash-faults by Alistarh, Attiya, Guerraoui and Travers [AAGT12], we show that the AA-based approximation phase, and thus our algorithm, requires only a constant number of rounds to converge when $n > t^2 + 2t$. Furthermore, in this case it also achieves tight namespace of size n, because our id selection scheme ensures that Byzantine processes are not able to introduce more than t identifiers.

Even in the favorable case above, the number of communication rounds can be impairment for time-constrained applications. Therefore, we then address the challenge of solving order-preserving renaming in as few communication rounds as possible. We show that, if the number of faults is known to be restricted to $n > 2t^2 + t$, order-preserving renaming can be solved in just 2 rounds. For this purpose, we present a new algorithm where renaming is done by having processes exchange their initial ids, perform one echoing round, and then count the numbers of echoes to calculate new names.

After solving order-preserving renaming efficiently for different values of $t < n/3$, we then ask a natural question whether it is possible to solve this problem for larger values of t (as it is the case for the original renaming problem). We answer this question in the negative by proving that it is impossible to solve order-preserving renaming if $t \geq n/3$, even if one could use randomized algorithms. Thus, randomization does not help solving this problem, in contrast to original renaming. To our knowledge, this is the first result that separates the resiliency of renaming and order-preserving renaming algorithms. Interestingly, the impossibility applies to a target namespace of any size. The proof reduces the case of $n > 3$ to the case of $n = 3$. It then considers a candidate algorithm for the case $n = 3$ and shows that it must fail, by constructing an indistinguishability ring of executions that violates the properties of order-preserving renaming. This technique was applied previously to the problem of consensus with Byzantine failures [FLM85]. Here we extend it to order-preserving renaming, which is a weaker problem and hence harder to prove impossibility results for. In our proof, we arrange processes in a ring of size larger than the target namespace and use the following argument to establish a contradiction: roughly, we show that names must increase as we traverse the ring in one direction, but this exhausts the target namespace after going around the ring.

1.3 Outline of the Thesis

In Chapter 2, we summarize the related work. In Chapter 3, we describe the model and state the problems considered in this thesis.

In Chapter 4, we address renaming in the crash fault model. We present a renaming algorithm for $t < n$, give the correctness proof and complexity analysis.

In Chapters 5 and 6, we consider the Byzantine fault model. In Chapter 5, we present renaming algorithms for $t < n$ under different adversaries, give the correctness proofs and complexity analysis. In Chapter 6, we address the order-preserving variant of the renaming problem. We present algorithms for different values of $t < n/3$, give the correctness proof and complexity analysis. We then prove that it is impossible to solve this problem if $t \geq n/3$.

In Chapter 7, we conclude the thesis and outline the directions for future work.

System Model				**Round Complexity**	**Reference**
Type of Faults	Algorithm	Adversary	Resiliency	**of Tight Renaming**	
Crashes	Deterministic	n/a	$t < n$	$\Theta(\log n)$	[CHT99]
	Randomized	adaptive	$t < n$	$O(\log\log n)$ whp	Chapter 4
Byzantine	Deterministic	n/a	$t < n/3$	$O(\log n)$	[OBG08] [a]
		n/a	$n/3 \leq t < n$	Θ(unbounded)	[OBG08]
	Randomized	non-rushing	$t < n$	$O(\log n)$ whp	Chapter 5, Section 5.1
		rushing	$t = 1$	$O(\log n)$ whp	Chapter 5, Section 5.2
		rushing, computationally bounded	$t < n$	$O(\log n)$ whp	Chapter 5, Section 5.2

Table 1.1: Round complexity of renaming in synchronous message-passing systems.

[a] This is the only algorithm in the table that does not solve *tight* renaming as its target namespace is of size $2n$.

System Model[a]		**Order-preserving Renaming**		**Reference**
Type of Faults	Resiliency	**Namespace Size**	**Round Complexity**	
Crashes	$t < n$	n	$\Theta(\log n)$	[Oku10]
	$t < O(\sqrt{n})$	n	$O(1)$	[AAGT12]
Byzantine	$t < n/3$	$n + t - 1$	$O(\log n)$	Chapter 6, Section 6.1
	$t < O(\sqrt{n})$	n	$O(1)$	Chapter 6, Section 6.2
	$t \geq n/3$	any	impossible [b]	Chapter 6, Section 6.4

Table 1.2: Round complexity of order-preserving renaming in synchronous message-passing systems.

[a] The results presented in the table concern deterministic algorithms. To our knowledge, no randomized algorithms have been proposed in the literature.
[b] The impossibility result also applies to randomized algorithms.

Chapter 2

Related Work

The renaming problem and its order-preserving variant were originally introduced by Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [ABND$^+$90] in the asynchronous message-passing model with crash failures. In this model consensus is known to be impossible [FLP85], so the authors of [ABND$^+$90] addressed the possibility of solving these new problems in the same setting. They show that *tight* renaming (i.e., renaming with the smallest namespace, of size n) is also impossible in the asynchronous model. The authors then present a renaming algorithm with $n > 2t$ and target namespace of size $n + t - 1$, and an order-preserving renaming algorithm with $n > 2t$ and target namespace of size $2^t(n - t + 1) - 1$. Both algorithms tolerate the optimal number of crashes and the bounds on the target namespace were also shown to be optimal [ABND$^+$90, HS99]. Subsequently, the renaming problem has been extensively studied in both message-passing and shared memory systems. From this point, we limit the related work discussion to synchronous message-passing systems considered in this thesis. We direct the interested reader to [BEW11, AACH$^+$14] for surveys on renaming in other models.

2.1 Renaming in Synchronous Systems

In synchronous systems, renaming can be solved using Reliable Broadcast [BT85, GT89] or consensus [LSP82b]: processes can simply agree on the set of all ids existing in the system and then, each process chooses as a new name the rank of its own id the the set. This approach requires linear round complexity [DS82]. However, intuitively the renaming problem is "weaker" than consensus. In fact, renaming is considered the simplest non-trivial distributed computing task [CHT99]. Therefore it is tempting to search for more efficient sublinear solutions than simply using consensus. Indeed, several such solutions were presented for different settings. Below we summarize the prior work on renaming and order-preserving renaming in synchronous systems prone to both crashes and Byzantine faults.

2.1.1 Renaming with Crash Faults

Tight renaming in the synchronous message-passing model is first studied by Chaudhuri, Herlihy, and Tuttle [CHT99]. The authors define *comparison-based* algorithms, which distinguish states only through comparisons [1]. In comparison-based algorithms, two processes, for which all comparisons on their state yield the same result, are in an *indistinguishable* state and will decide on the same result. The authors show that such algorithms are vulnerable to a "sandwich" failure pattern, which forces processes to continue in indistinguishable states for $\Omega(\log n)$ rounds. The proof argument goes as follows. Let m be a number such that $n = 3m + 1$ and assume that in each round processes exchange all the information available to them. In the first round, the "sandwich" failure pattern causes m processes $\{p_1, \ldots, p_m\}$ with the lowest ids and m processes $\{p_{2m+2}, \ldots, p_{3m+1}\}$ with the highest ids to fail in the following way: each surviving process $p_{m+j} \in \{p_{m+1}, \ldots, p_{2m+1}\}$ receives messages only from processes $\{p_j, \ldots, p_{2m+j}\}$. As a result, each surviving process receives $2m + 1$ messages, and sees its rank in the set of all processes as $m + 1$. In the following round, the same happens to the two thirds of the surviving processes and so on. The "sandwich" failure pattern can continue for $\Omega(\log n)$ rounds, keeping the surviving processes in indistinguishable states. This yields an $\Omega(\log n)$-round lower bound for deterministic comparison-based algorithms.

[1]For example, in a comparison-based algorithm, processes cannot use values of their ids to decide in which round to send a message.

The authors match the lower bound via an elegant algorithm [CHT99] for any $t < n$ that works as follows. A process chooses a new name by selecting one bit at a time, starting with the high-order bit and working down to the low-order bit. In each round processes exchange their ids and the intervals of new names in which they are interested. Then, processes split the received ids in two sets, choosing 0 if their own id belongs to the first half, or 1 otherwise, and repeat the procedure. Since each bit of a new name is chosen in a separate round, the round complexity of the algorithm is $\Theta(\log n)$.

2.1.2 Renaming with Byzantine Faults

The first paper to address the renaming problem in systems prone to Byzantine failures is [OBG08] where Okun, Barak, and Gafni show that renaming can be solved deterministically for any $t < n$ with the following algorithm: a process waits until the round corresponding to the value of its id, then picks an available name, and sends its decision to other processes (this algorithm is *not* comparison-based). The remaining processes exclude the announced decision from the available names. Since all correct processes have different ids, no two correct processes decide in the same round, and hence they always choose distinct names. This algorithm, however, has an unbounded running time because its running time depends on the values of original ids, which can be taken from an *unbounded* namespace. The same paper shows that the unbounded running time is unavoidable for deterministic renaming if $t \geq n/3$.

Finally, the paper presents a renaming algorithm for $t < n/3$ with round complexity of $O(\log n)$. The solution is based on the crash-tolerant renaming algorithm introduced in [CHT99] and discussed in the previous section. More precisely, the algorithm in [OBG08] applies a version of the automatic crash-to-byzantine translation techniques to the algorithm in [CHT99]. The translation techniques, introduced by Neiger and Toueg [NT88] and Bazzi and Neiger [BN01], are based on the following idea. Processes run a crash-tolerant algorithm and, in parallel, echo all messages received in each round. Roughly, if a message is eventually echoed by sufficiently many, $n - t$, processes, then the message is *accepted* and used in the original crash-tolerant algorithm. Otherwise, the sender of that message is considered faulty and is excluded from further communication. Note that this procedure assumes that each message carries the sender's id and the sender cannot lie about its id. Thus the translation techniques of [NT88, BN01] require that the initial ids of processes are known to every process *a priori*. Eliminating this assumption is the main difficulty in adapting the translation techniques to the renaming problem. Note that at least $n - t$ correct processes always echo each others' messages correctly, so their messages are always accepted by all correct processes. However, the correct processes must deal with the cases when the same Byzantine process announces different ids to different correct processes. Instead of discarding messages with Byzantine ids, correct processes may have to continue echoing them to ensure the same decision is made by all correct processes to either accept or discard the message. As a result, the transformed algorithm in [OBG08] may deal with more than n ids and can no longer ensure the tight namespace; its target namespace increased to $2n$.

Since the crash-to-byzantine translation techniques work for $t < n/3$ and have constant round complexity overhead per translated round, the resulting transformed algorithm also tolerates up to $t < n/3$ Byzantine failures and it has round complexity $O(\log n)$ rounds.

2.1.3 Order-Preserving Renaming with Crash Faults

Prior to the results produced in this thesis, we were not aware of any results on order-preserving renaming in the Byzantine setting. Therefore, in this section we only discuss the existing results on order-preserving renaming in systems prone to crashes.

The first crash-tolerant algorithm that solves this problem was presented by Okun [Oku10]. The author finds a novel connection between order-preserving renaming and the *approximate agreement* problem, and shows that this approach has round complexity of $O(\log n)$. In approximate agreement (AA), processes, starting with real values as inputs, are required to decide on values within a bounded range from each other. The algorithm in [Oku10] uses AA to solve order-preserving renaming as follows. First, processes exchange their ids, sort the received ids, and propose a new name for each id based on its rank in the sorted list. Due to crashes, processes may have received different sets of ids and therefore may propose different names for the same id. These discrepancies are later reduced by running (in parallel) AA for each id: in every round, each process sends the candidate name for each id, received the candidate names from other processes, and updates each value by averaging all the values received for that id. After the approximate agreement ensures that the values for the same id are within a safe distance from each other, each process decides based on the output of AA for its own id. Since the initial discrepancies in the proposed names are at most n, the approximate agreement requires at most $O(\log n)$ rounds to bring the values within a small distance. Thus, the round complexity of the algorithm is $O(\log n)$.

Recently, Alistarh, Attiya, Guerraoui and Travers [AAGT12] made the algorithm of Okun [Oku10] *early terminating*, i.e. the time complexity of the algorithm depends on f, the number of actual faults occurred in a given execution. The authors of [AAGT12] observed that, in fact the initial discrepancies in the proposed names are at most f in an execution with f faults. Hence, the approximate agreement converges in only $O(\log f)$ rounds instead of $O(\log n)$. Interestingly, the authors also observed that the algorithm can decide in constant number of rounds if the number of actual faults is bounded by $n > 2f^2$. This is because in that case approximate agreement converges in a constant number of iterations.

2.1.4 Discussion

The surveyed results show that there are settings, in which renaming can be solved efficiently without resorting to costly consensus-based solutions. In particular, it was shown that renaming can be solved in $O(\log n)$ rounds with up to $t < n$ crashes or $t < n/3$ Byzantine faults. However, many interesting questions remained unanswered. For example, the literature does not address randomized solutions for renaming. But we know from other settings that randomization is a powerful technique in circumventing impossibility results [BO83] and outperforming optimal deterministic algorithms [Bra87]. Therefore, studying randomized solutions for renaming is essential for a more complete understanding of this problem. In particular, it is interesting to investigate if randomization allows circumventing the deterministic lower bounds—$\Omega(\log n)$ for $t < n$ crash faults and $\Omega(\text{unbounded})$ for $t < n$ Byzantine faults—on the round complexity of renaming algorithms.

Other open problems concern the order-preserving variant of renaming. In particular, this problem had not been solved in the Byzantine setting. This thesis also addresses this open problem.

2.2 Other Related Problems

As we have stated above, the thesis will address the use of randomization in solving renaming. Randomization techniques have been used in other problems that are conceptually similar to renaming. In the following, we discuss the related results.

2.2.1 Balls into Bins

Tight renaming can be seen as a balls-into-bins load balancing problem, where n balls (processes) must be randomly placed into n distinct bins (names). Early work on balls-into-bins problem addressed a scenario in which balls are placed into bins by trying randomly in sequential order, e.g. [Gon81]. Since then, the problem has been extensively studied in different models, e.g. [ACMR95, BKSS13, Mit01]. In particular, the closest model to ours is the one where balls are placed by contacting bins in parallel. If, in some round, more than one ball contacts the same bin, the bin decides which ball to accept. The accepted balls are placed into respective bins, the rejected balls continue retrying other bins in the following rounds until they are accepted.

Parallel balls-into-bins problem is motivated by distributed load-balancing with bandwidth limitations. Therefore, in most existing solutions balls contact a limited number of bins at once. It is easy to show that the naive approach, in which each ball contacts one random bin at a time and retries if it is not accepted, requires $\Omega(\log n)$ rounds whp. The naive strategy can be improved by "the power of two choices" technique (generalized to d choices). In this technique, each ball contacts $d \geq 2$ random bins in parallel and, if it is accepted to more than one bin, the ball chooses one of those bins. If the ball is not accepted by any bin, it retries. This approach was shown to require $\Omega(\sqrt{\log n})$ rounds whp [ACMR95].

Several algorithms have been proposed for this setting, e.g. [BKSS13, LW11b], which achieve a better, sub-logarithmic complexity. Unfortunately, none of these solutions can be used to obtain tight renaming with up to $t < n$ crashes. This is because they either relax the exact one-ball-per-bin requirement [LW11b], or require balls to always have consistent views when making their choices (which cannot be guaranteed under crash faults). For a complete survey of existing results on parallel load-balancing, we refer the reader to [LW11a].

2.2.2 Randomized Symmetry Breaking

The randomization techniques employed in this thesis are similar to the techniques used for breaking symmetry in distributed systems. One way of breaking symmetry is by electing a unique *leader*. Several papers applied randomization techniques to the leader election problem, e.g. [FL87, IR81, KPP+13]. A notable example in this line of research is the work by Itai and Rodeh [IR81] that addresses symmetry breaking in anonymous rings. The paper presents a randomized ring leader election algorithm that works as follows. Each process randomly picks a value from a given range and sends it through the ring. The process that has picked the largest value becomes a leader. In case of a collision, the processes with the largest value repeat the algorithm until one process wins. Though (as we will see in the following chapters) our techniques are similar in flavor, these related results solve a different problem and cannot be directly translated into our setting.

Chapter 3

Model and Definitions

In this chapter we describe the system model and formally define the main problems addressed in the thesis.

3.1 Basic System Model

We consider a standard synchronous round-based message-passing system of $n \geq 3$ processes $p_1, \ldots, p_n$, where n is known a priori. Each process has a unique identifier, originally known only to the process itself.

We consider a fully-connected network with bidirectional links between every pair of processes (the communication links are assumed to be reliable). The bidirectional links of each process are numbered $1, \ldots, n$; there is no global assignment of processes to link numbers, and different processes may have inconsistent link numberings. We denote by $link_i(p_j)$ the number at p_i of the bidirectional link to p_j. At each process, we assume that link number n is the self-loop, and that a process knows the number of the link from which it receives a message.

Execution proceeds in rounds; in each round, a process can send messages to its neighbors, receive at most one message from each neighbor, and change state.

In randomized algorithms, the process obtains a fixed-length string of private random bits before changing state.

3.2 Fault Models

Throughout the thesis, we assume different models of failures. We say that the failures are controlled by an adversary. In the following we describe different adversaries that we consider.

3.2.1 Crash Faults

In Chapter 4 we assume that processes may crash. The adversary can make any process crash at any moment in the execution. We assume that the adversary knows the protocol and can observe the random choices of all processes; thus, the adversary may decide which processes fail based on the random choices of all processes (such adversary is often called *adaptive*). We assume that up to t processes may crash, for any $t < n$. Moreover, we assume that the crashed processes stop executing and do not recover.

3.2.2 Byzantine Faults

In Chapter 5 and Chapter 6 we assume a more powerful adversary that can corrupt up to t processes; the corrupted processes, which we call *Byzantine* [LSP82a], may exhibit arbitrary behavior and are allowed to collude among themselves. We assume that the adversary knows the protocol, can decide which processes to corrupt at any time during the execution, and has full control of the corrupted processes. Here, we assume that the adversary does *not* have access to the private random bits of the correct processes, and we allow the adversary to corrupt at most t processes. For each of our algorithms, we will specify the upper bounds on t.

Since some of our algorithms are randomized, we further need to specify when the adversary is allowed to learn the random choices of the correct processes. In particular, we distinguish between *rushing* and *non-rushing* adversaries.

Non-rushing Adversary. Under the non-rushing adversary, in each round all processes send messages simultaneously. Thus, the non-rushing adversary cannot decide the messages to be sent during a particular round based on the messages the corrupted processes receive from correct processes during that same round.

Rushing Adversary. Under the rushing adversary, it is pessimistically assumed that the messages addressed to the Byzantine processes are always delivered immediately, and that the adversary has time to inspect the messages addressed to the corrupted processes before issuing its messages in the same round.

The adversary is allowed to eavesdrop the communication between the correct processes. However, the adversary is unable to send messages in the name of correct processes, or modify messages that the correct processes send.

3.3 Cryptographic Techniques

When we use cryptography in Section 5.2 of Chapter 5, we consider a computationally bounded adversary that is limited to computing polynomially bounded functions. We assume the availability of a cryptographic commitment scheme. Roughly speaking, cryptographic commitment allows processes to commit to a value without revealing it and then reveal a value when required. Such a scheme has two separate stages, commitment stage and revealing stage, with the following interface for each respective stage:

- COMMIT(*value*): a process generates a cryptographic commitment to a value, without revealing the value.
- REVEAL(*value*): a process reveals the value, which must match the previously announced commitment.

The properties of a commitment scheme can be informally stated as follows.

- *Hiding:* it is computationally hard for the receiver to know in the commitment stage the value to which the sender commits.
- *Binding:* it is computationally hard for the sender to commit more than one value and to reveal a value that it has not committed.

For a more formal treatment, we refer the reader to [Gol01].

3.4 Problem Statement

We now present the definitions of the main problems considered in the thesis.

3.4.1 Renaming

The renaming problem can be precisely defined as follows.

Each process has an *initial id* in some *original namespace*, and it must decide on a new name in some *target namespace* $\{1, \ldots, M\}$, where $M \geq n$ is a parameter of the problem, such that the following conditions must hold [ABND+90]:

- *Termination:* Each correct process eventually decides on a new name.
- *Validity:* If a correct process decides, it decides on a new name in $\{1, \ldots, M\}$.
- *Uniqueness:* No two correct processes decide on the same new name.

For randomized algorithms, the termination property is weakened to

- *Termination with probability* 1*:* With probability 1, every correct process eventually decides on a new name.

When $M = n$ the problem is called *tight* renaming.

3.4.2 Order-Preserving Renaming

The stronger *order-preserving* variant of renaming is obtained by adding the following property:

- *Order-preserving property [ABND$^+$90]:* New names of the correct processes preserve the order of the initial ids.

3.5 Complexity Measures

We are mainly interested in the round complexity of our algorithms, i.e., the total number of rounds required to all correct processes terminate. For completeness, we will also discuss the message complexity (the total number of messages sent by the processes in an execution) and bit complexity (the total number of bits per message). For randomized algorithms, the round complexity and message complexity are required to hold with high probability (whp), i.e., with probability at least $1 - 1/n^c$, where c is an independent constant.

In general, algorithms can have unbounded running time. In this work we are interested in *efficient* algorithms, whose round complexity can be bounded by a function of n, the number of processes (this property is also known in the literature as *strong termination* [Dij82]). For randomized algorithms, we require the *probabilistic* round complexity be bounded by a function of n.

Chapter 4

Renaming with Crash Faults

In this chapter we assume that processes may fail by crashing and up to $t < n$ processes may crash. In this model, the efficiency of deterministic algorithms is constrained by the $\Omega(\log n)$ lower bound on the round complexity. Therefore, to achieve better results we resort to randomization.

As we have previously discussed, randomized renaming is related to the balls-into-bins technique. Extending this classical technique, we propose *Balls-into-Leaves*, a randomized algorithm that places n balls into n leaves of a binary tree in $O(\log\log n)$ rounds whp. At a high level, our algorithm works as follows: processes (balls) start at the root of a virtual binary tree, whose n leaves correspond to the target names. In each iteration, each ball picks a random available leaf and broadcasts its choice; in case of collisions, a deterministic rule is used to select a winner. The remaining balls backtrack towards the root, stopping at the lowest node at which the ball can still find an available leaf within the corresponding subtree. (Please see Figure 4.2 for an illustration.) Balls then iterate this procedure, exchanging information and progressively descending in the tree. We then extend our algorithm to offer early termination in $O(\log\log f)$ rounds whp, where f is the number of actual failures in a given execution.

4.1 Balls-into-Leaves Algorithm

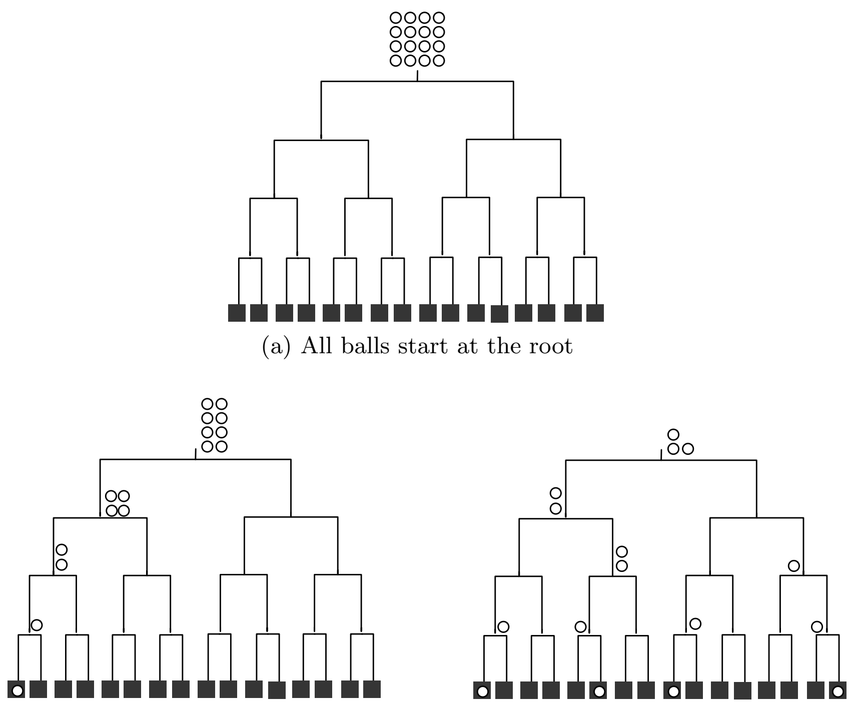
In this section, we describe in more detail our Balls-into-Leaves algorithm.

The algorithm treats processes as *balls* and target names as *bins*, where each ball wants to find an exclusive bin for it. The algorithm organizes the n bins as leaves of a binary tree of depth $\log n$[1]. Balls have unique labels (the processes' initial ids), and they can communicate by broadcasting messages.

The algorithm at a high level. Each ball starts at the root of the tree and descends the tree along a random path. As balls descend, they communicate with each other to determine if there are collisions. Collisions occur if many balls try to go to the same leaf or, more generally, if many balls try to enter a subtree without enough capacity for them. For example, if all n balls at the root tried to enter the left subtree, they would collide since the subtree has capacity for only $n/2$ balls. When balls collide, the algorithm assigns priorities to them based in part on their unique label; balls with higher priorities keep descending, while the others stop. Because balls pick random paths, very few collide at higher levels, so balls quickly descend the tree and soon after find an exclusive leaf for them.

Local tree, candidate path, remaining capacity. The binary tree has $\log n$ levels. To finish in $O(\log \log n)$ rounds whp, balls must descend many levels with a single round of communication. To do so, each ball b_i keeps a *local tree*, containing the current position of each ball, including itself. Initially, all balls in the local tree of b_i are at the root (Fig. 4.1(a)). In a single round, a ball b_i picks a random *candidate path* in its local tree: starting with its current position, b_i successively chooses the left or right subtree to follow for each level, until the leaf is reached. The choice between left and right subtree is weighted by the *remaining capacity* of each subtree (within b_i's local tree). The remaining capacity is the number of leafs of the subtree minus the number of balls in the subtree. Say, if one subtree has no remaining capacity, b_i chooses the other with probability 1. In

[1]To simplify exposition, we assume n is a power of two.


(a) All balls start at the root

(b) All balls choose the first leaf

(c) Choices are well distributed

Figure 4.1: One phase of execution

this way, b_i picks the entire candidate path to the leaf *locally*, without communication with other balls and without regard to collisions. Ball b_i does not yet go down its path (this will happen later). Rather, b_i broadcasts its path and waits for the paths of others; this requires a round of communication.

Collisions, priority. Once ball b_i has received the candidate paths of other balls, b_i can calculate new positions of these balls. Ideally, a ball just follows its candidate path. But b_i may find that candidate paths collide: more balls may try to enter a subtree than the subtree's remaining capacity. In this case, b_i allows balls with higher priority to proceed, while others must stop where the collision occurs (and the rest of their candidate path is discarded). The priority is determined by an order $<_R$ where smaller balls under $<_R$ have higher priority.

Definition 1 (Priority Order $<_R$)**.** *Let η_i and η_j be the current nodes of balls b_i and b_j. Then,*

$$b_i <_R b_j \iff (depth(\eta_i) > depth(\eta_j)) \vee ((depth(\eta_i) = depth(\eta_i)) \wedge (b_i < b_j)) .$$

Under $<_R$, balls are ordered by their depth in the tree (balls downstream ordered first), breaking ties by their unique labels. To implement these priorities, b_i iterates over all balls $\bar{b}$ in $<_R$ order; for each $\bar{b}$, b_i lets $\bar{b}$ follow its candidate path until $\bar{b}$ reaches a *full* subtree—one with no remaining capacity. If $\bar{b}$ is lucky, it ends up at a leaf in b_i's local tree; otherwise, it stops at a higher level. Irrespective of where $\bar{b}$ stops, the algorithm ensures that there is enough space below to accommodate it. Because balls lower in the tree have a higher priority, this space cannot be displaced subsequently by balls higher in the tree. Figure 4.1 shows the local tree before and after new positions are calculated.

Failures, synchronization, termination. A ball may crash while broadcasting its candidate path; some balls may receive this broadcast, while others do not. The result is that the local tree of balls may diverge. To resynchronize, after b_i has updated its local tree using the candidate path, b_i broadcasts its current position, and waits to learn the position of other balls; this requires a round of communication. Based on the received information, if necessary b_i corrects the positions of balls in its local tree; if b_i does not hear from a ball in its tree, b_i removes it (the ball has crashed). If b_i finds that every ball is at a leaf, it terminates. Otherwise, it picks a new candidate path for itself and repeats the entire procedure.

Detailed pseudocode. Algorithm 1 describes the data structures and functions and Algorithm 2 gives the detailed pseudocode. Initially, each ball b_i broadcasts its label, receives labels from other balls, and inserts them at the root of its local tree (Line 1). Then, b_i repeatedly executes the main loop (Lines 2–29); each iteration is called a *phase*, and each phase has two communication rounds. In round one, b_i first chooses its candidate path (Lines 5–10) one edge at a time, where the probability of each child is the ratio of remaining slots in either subtree (Line 6). Then b_i broadcasts its path (Line 11). After receiving the paths of others, b_i iterates over the balls $\bar{b}$ in $<_R$ order, to compute their new positions. Each ball $\bar{b}$ moves down its path as long as the subtree has remaining capacity (Lines 12–18). Balls that do not announce their paths have crashed and are removed from the local tree (Lines 19–20). In the second round (Lines 22–28), b_i sends its position, receives positions of other balls, and updates its local tree, again removing balls which fail to send their positions. If all balls in the local tree have reached leaves, b_i terminates. It is easy to change the algorithm to allow a ball to terminate as soon as it reaches a leaf. Such modification requires additional checks that have been left out in favor of clarity.

Algorithm 1 Balls-into-Leaves Algorithm

Data Structures and Functions

Data Structures

- binary tree with n leaf leaves;
- path$_i$: an ordered set of nodes;

Operations over the tree:

- Remove(b_j) removes b_j from the tree;
- CurrentNode(b_j): current node of b_j;
- UpdateNode(b_j,η): removes b_j from its current node and places it at node η;
- OrderedBalls() returns a set of all balls in the tree, ordered by $<_R$ (first by their depth in the tree, then by ids);
- RemainingCapacity(η): number of leaves in the subtree rooted at node η minus number of balls in that subtree;
- η.LeftChild(), η.RightChild(), η.isLeaf() are operation over nodes of the tree.

Additional Functions

- First(): first element in a set;
- Next(): iterator over a set (returns the next element in a set, advancing to the next position);
- RandomCoin(p): returns *heads* with prob. p, or *tails* with prob. $(1-p)$.

Algorithm 2 Balls-into-Leaves Algorithm (continued)

Code for Ball b_i

```
 1: Initialize: broadcast $\langle b_i \rangle$; $\forall b_j$ received: insert $b_j$ at the root;
 2: repeat                                              ▷ begin Phase $\phi \leftarrow 1, 2, 3, \ldots$
 3:     $\eta \leftarrow$ CurrentNode($b_i$);                          ▷ begin Round 1 of Phase $\phi$
 4:     path$_i \leftarrow \{\eta\}$;
 5:     while not $\eta$.IsLeaf() do                                 ▷ choose path randomly
 6:         $coin \leftarrow$ RandomCoin($\frac{\text{RemainingCapacity}(\eta.\text{LeftChild}())}{\text{RemainingCapacity}(\eta)}$);
 7:         if $coin = heads$ then $\eta \leftarrow \eta$.LeftChild();
 8:         else $\eta \leftarrow \eta$.RightChild();
 9:         path$_i \leftarrow$ path$_i \sqcup \{\eta\}$;
10:     end while
11:     broadcast $\langle b_i$,path$_i\rangle$;                               ▷ exchange paths
12:     for all $b_j \in$ OrderedBalls() do
13:         if $\langle b_j$,path$_j\rangle$ has been received then    ▷ move balls in the priority order
14:             $\eta \leftarrow$ path$_j$.First();
15:             while remainingCapacity($\eta$)$> 0$ do
16:                 $\eta \leftarrow$ path$_j$.Next();
17:             end while
18:             UpdateNode($b_j$,$\eta$);
19:         else
20:             Remove($b_j$);
21:     end for                                          ▷ begin Round 2 of Phase $\phi$
22:     broadcast $\langle b_i$, CurrentNode($b_i$)$\rangle$;                       ▷ synchronize
23:     for all $b_j \in$ OrderedBalls() do
24:         if $\langle b_j$,$\eta_j\rangle$ has been received then
25:             UpdateNode($b_j$ ,$\eta_j$);
26:         else
27:             Remove($b_j$);
28:     end for
29: until $\forall b_j \in$ OrderedBalls(): CurrentNode($b_j$).IsLeaf();
```

4.1.1 Tight Renaming using Balls-into-Leaves

We now prove that the Balls-into-Leaves algorithm solves tight renaming in $O(\log \log n)$ communication rounds whp. The process with original identifier id_i runs Balls-into-Leaves for the ball labeled id_i. It then returns the (left-to-right) index of the leaf where the ball terminates, and outputs this rank as its name.

Name uniqueness follows from the fact that no two correct balls can terminate on the same leaf (Theorem 5). *Validity* follows because the number of leaves is n. Termination and complexity follow from the complexity analysis of the Balls-into-Leaves algorithm.

4.1.2 Correctness

We now prove that in the Balls into Leaves algorithm correct balls terminate at distinct leaves.

We first notice that, at the beginning of each phase, the positions of all correct balls are synchronized across the views. Positions are considered at the beginning of a phase.

Proposition 1. *For any phase $\phi \geq 1$, if balls b_i and b_j are correct, then the tree position of b_i at ball b_i is the same as the position of b_i at ball b_j.*

Proof We proceed by induction on the phase index. At the beginning of phase 1, this holds since all n correct balls have broadcasted their labels and have been placed at the root in local views of all correct balls (Line 1).

For the induction step, assume by contradiction that the claim holds in $\phi \geq 1$, but not in $\phi + 1$. This is only possible if b_j has not sent its position to b_i in ϕ (Line 22 of Algorithm 2). However, since b_j is correct, b_j must have executed Line 22 in phase ϕ. Contradiction. □

This implies the following.

Proposition 2. *For any phase $\phi \geq 1$ and any local view, the number of correct balls in each subtree is less or equal to the total number of balls in that subtree.*

The priority order $<_R$ ensures that the descent of correct balls is simulated consistently across views.

Proposition 3. *For any phase $\phi \geq 1$, if b_i and b_j are correct, then either $b_i <_R b_j$ in every view, or $b_i >_R b_j$ in every view.*

Proof By Proposition 1, the position of correct balls is synchronized across the views. Assume without loss of generality that $b_i < b_j$. Thus, by definition of the order $<_R$, $b_i <_R b_j$ in each view. □

Lemma 4. *For any phase $\phi \geq 1$, in any local view, the number of correct balls in each subtree is less or equal to the number of leaves in that subtree.*

Proof We prove the claim by induction over the phase index ϕ. For $\phi = 1$, the claim holds since all n balls are at the root of the tree.

Assume the claim is true for $\phi \geq 1$. Since each subtree contains at all least correct balls (Proposition 2), and (Proposition 3), balls simulate the descent of correct balls in a consistent order (if $b_j <_R b_k$, b_j is moved before b_k), when b_i simulates the descent of each ball locally, every ball (including b_i) stops in a subtree where it still fits among at least all correct balls. □

We now prove that the algorithm ensures that the balls terminate at distinct leaves.

Theorem 5. *In the Balls-into-Leaves algorithm, correct balls terminate at distinct leaves.*

Proof From Proposition 1 and Lemma 4, after correct ball b_i reaches a leaf and announces its position, all correct balls have b_j at its leaf in their local views and thus never propose a path to that leaf. By algorithmic construction, balls do not move once they have reached the bottom. □

Finite Deterministic Termination

In the following, we show that the algorithm terminates always in a finite number of rounds.

Lemma 6. *If no failures occur in some phase ϕ, at least one ball at an inner node reaches a leaf in its local view of the tree.*

Proof Let b_j be the highest priority correct ball among the balls at inner nodes (not leaves) at the beginning of ϕ. By algorithmic construction, b_j chooses in Round 1 a path to an empty leaf in its local view of the tree. Since by assumption no crashes occur, balls receive identical sets of paths and move balls down in the same order. By algorithmic construction, balls at the leaves are not moved. So, b_j is the first to move down in its own view and thus it will reach the leaf chosen by its path. □

Since by Lemma 6 processes require at most n fault-free phases to reach the bottom, and there are at most $t < n$ faults in total, the algorithm terminates in $O(n)$ phases deterministically.

4.2 Complexity Analysis

In this section, we prove that Balls-into-Leaves terminates in $O(\log \log n)$ rounds with high probability.

For clarity, we first consider a failure-free execution. We then show that faults do not slow down the progress of the protocol. (Intuitively, collisions are less likely as the number of surviving balls decreases).

Without crashes, local views of the tree are always identical, and we therefore focus on one local view. The analysis consists of two parts. In the first part, we show that, after $O(\log \log n)$ phases, the number of balls at each node decreases to $O(\log^2 n)$. In the second part, we consider an arbitrary path from the root to the parent of a leaf. We use the fact that there are $O(\log^2 n)$ balls at each node, and show that the path becomes completely empty during the next $O(\log \log n)$ rounds, with high probability. By a union bound over all such paths, we obtain that the tree is empty whp.

We first prove useful invariant of Balls-into-Leaves algorithm that we call *Path Isolation Property*. Informally stated, this says that no new balls appear on any path from the root.

Lemma 7 (Path Isolation Property). *For any phase $\phi \geq 1$ and path π from the root, the set of balls on π in ϕ is a superset of balls on π in $\phi + 1$.*

Proof By construction, balls can only move down the tree. Fix some node η on path π. The only way new ball b_i at some node μ in phase ϕ reaches η in phase $\phi + 1$ is by constructing in ϕ a path from μ that contains η. Thus, by construction of a binary tree, μ is on π. $\square$

We use some notation and facts from theory of probability, shown in Figure 4.2.

4.2.1 Bounding the Number of Balls at a Node

We now give a lower bound on how fast the number of balls at each node decreases. Consider some node η, and let pM and $(1-p)M$ be the remaining capacities of its left and right subtrees, respectively, for some integer M, and $0 \leq p \leq 1$ in phase ϕ. By construction, at most M balls reach η. The balls that get stuck at η in phase ϕ+1 are those which had η on their paths in ϕ, but did not fit in a subtree below η. By Lemma 4, these balls have enough space in the sibling subtree of η. We use the notation $balls(\eta, \phi)$ to denote the number of balls at node η in phase ϕ. We denote by $b_{max}(\phi)$ the most populated node in phase ϕ.

Lemma 8. *For some node η, let pM and $(1-p)M$ denote remaining capacities of its left and right subtrees respectively, in some phase ϕ. If $M' \leq M$ balls choose between the left and right subtrees of η in ϕ, then, for any integer $x > 0$, $\Pr(balls(\eta, \phi + 1) > x) \leq \Pr(|\frac{M}{2} - \mathrm{B}(M, \frac{1}{2})| > x)$.*

Proof Choosing between the left and right subtrees of η can be seen as choosing between two bins with capacities pM and $(1-p)M$. Each of the $M' \leq M$ balls chooses independently between the two bins with probabilities p and $(1-p)$ respectively. If there is space in the chosen bin, then the ball is accepted; if the bin is full, then the ball is rejected.

$B(n,p)$ is the binomial distribution with parameters n and p. By abuse of notation, $B(n,p)$ also denotes a random variable that follows the binomial distribution.

Fact 1. *If* $X \sim \mathrm{B}(M,p)$*,* $Y \sim \mathrm{B}(M',p)$*, such that* $\mathrm{E}[X]$ *and* $\mathrm{E}[Y]$ *are integers, and* $M \leq M'$*, then*

$$\Pr(|\mathrm{E}[X] - X| > x) \leq \Pr(|\mathrm{E}[Y] - Y| > x).$$

Fact 2. *If* $0 \leq p \leq 1$*,* $X \sim \mathrm{B}(M,p)$*,* $Y \sim \mathrm{B}(M,\frac{1}{2})$*, such that* $\mathrm{E}[X]$ *and* $\mathrm{E}[Y]$ *are integers, then*

$$\Pr(|\mathrm{E}[X] - X| > x) \leq \Pr(|\mathrm{E}[Y] - Y)| > x).$$

Intuition. Intuitively, the probability density function of $\mathrm{B}(n,p)$ is *least* concentrated around the mean for $p = \frac{1}{2}$ (recall that the standard deviation of $\mathrm{B}(n,p)$ is $\sqrt{np(1-p)} \leq \sqrt{n/4}$) .

Fact 3 (Chernoff Bound)**.** *If* $X \sim \mathrm{B}(m,p)$*, then* $\Pr(|\mathrm{E}[X] - X| > x) < e^{-\frac{x^2}{2mp(1-p)}}$.

Figure 4.2: Notation and facts from probability theory used in the proofs.

Since $M' \leq M$, there are three cases: either both bins are filled (perfect split), no bin has been filled, or exactly one bin has been filled, and there is some overflow. Note that, in the first two cases, η is empty in phase $\phi + 1$.

If one bin has filled, then, w.l.o.g., assume it is the left bin. Let Y be a random variable that counts the number of balls that have chosen left. Clearly, $Y \sim \mathrm{B}(M', p)$. Then, the number of rejected balls is $Y - Mp$. Since $Mp \geq M'p = E[Y]$, $Y - Mp \leq Y - E[Y]$.

From Fact 2, we obtain that $\Pr(|Y - E[Y]| > x) \leq \Pr(|\frac{M}{2} - \mathrm{B}(M, \frac{1}{2})| > x)$. □

Let us now consider what happens after the first phase.

Lemma 9. *Let i be the depth of node η. Then, for some constant $c > 0$, we have that*

$$\Pr\left(balls(\eta, 2) > c(\tfrac{n}{2^i}\log n)^{\frac{1}{2}}\right) < \tfrac{1}{n^c}.$$

Proof Initially n balls start at the root.

The initial capacity of the subtree rooted at η is $\frac{n}{2^i}$. Then, at most $\frac{n}{2^i}$ balls reach η.

Define $Y_\eta \sim \mathrm{B}(\frac{n}{2^i}, \frac{1}{2})$.

Applying Lemma 8 and the Chernoff bound (Fact 3),

$$\Pr(balls(\eta, 2) > x) < \Pr\left(|E[Y_\eta] - Y_\eta| > c(\frac{n}{2^i}\log n)^{\frac{1}{2}}\right) < \frac{1}{n^c}.$$

□

The analysis of the next phases is more involved, since we do not know the exact remaining capacities of each subtree. Therefore, we consider the worst case scenario by assuming that any node η has enough capacity to accommodate all balls on the path from the root to η.

Lemma 10. *For any $\phi > 1$, node η, and some const. $c > 0$,*

$$\Pr\left(balls(\eta, \phi + 1) > c(b_{max}(\phi))^{\frac{1}{2}}\log n\right) < \frac{1}{n^c}.$$

Proof By the path isolation property (Lemma 7), the only balls that may attempt to choose between subtrees of η are those on the path from the root to η. Let i be the depth of η. The total number of balls on path π is at most $i \times b_{max}(\phi)$. By Fact 1, the probability to have more rejected balls is the highest if we inflate the remaining capacity of the subtrees of η to $i \times b_{max}(\phi)$.

Thus, by Lemma 8, the probability that at least x balls get stuck at η can be bounded as follows,

$$\Pr(balls(\eta, \phi + 1) > x) \leq \Pr\left(\left|\frac{i \times b_{max}(\phi)}{2} - \mathrm{B}(i \times b_{max}(\phi), \frac{1}{2})\right| > c\right).$$

By the Chernoff Bound (Fact 3),

$$\Pr\left(\left|\frac{i \times b_{max}(\phi)}{2} - \mathrm{B}(i \times b_{max}(\phi), \frac{1}{2})\right| > c(i \times b_{max}(\phi)\log n)^{\frac{1}{2}}\right) < \frac{1}{n^c}.$$

Since $i \leq \log n$, the claim follows. □

Lemma 11. *For some constant $c' > 0$, after $O(\log\log n)$ phases,* $\Pr(balls(\eta, \phi) > O(1)\log^2 n) < \frac{1}{n^{c'}}$.

Proof Fix some constant $c > 0$. By Lemma 9, $\Pr\left(balls(\eta, 2) > c(n\log n)^{\frac{1}{2}}\right) < \frac{1}{n^c}$. By Lemma 10, $\Pr\left(balls(\eta, \phi+1) > c(b_{max}(\phi))^{\frac{1}{2}}\log n\right) < \frac{1}{n^c}$. Since $n^{\frac{1}{2}^{\log\log n}} = O(1)$, we pick some constant c_2 such that $n^{\frac{1}{2}^{c_2\log\log n}} = 1$.

Let $f(x) = x^{\frac{1}{2}}c\log n$. Taking $x = c(n\log n)^{\frac{1}{2}}$, $f^{c_2\log\log n}(x) = c^2\log^2 n$.

Applying Lemma 9, and then Lemma 10 iteratively for $c_2\log\log n$ phases, we obtain that

$$\Pr\left(b_{max}(\eta, c_2\log\log n) > c^2\log^2 n\right) < \frac{c_2\log\log c}{n^c}.$$

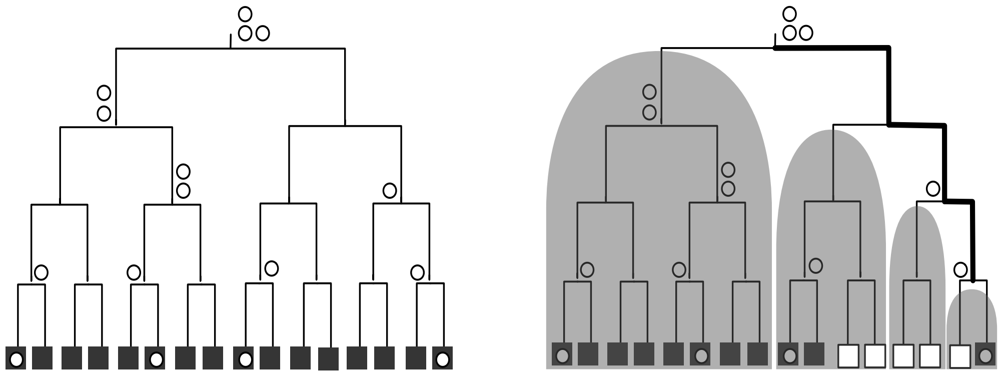
Therefore, there exists some small const. $\epsilon < 1$, such that $\frac{c_2\log\log n}{n^c} < \frac{1}{n^{(c-\epsilon)}}$, and the claim follows. □

4.2.2 Bounding the Number of Balls on a Path

Lemma 11 shows that, after $O(\log\log n)$ phases, the number of balls on each path is at most $O(\log^3 n)$ whp. In the following, we complete the argument by showing that all inner nodes of the tree are empty after another $O(\log\log n)$ phases.

To show this, instead of looking at nodes, we focus on paths from the root to a parent of a leaf (there are $n/2$ such paths). By the path isolation property (Lemma 7), new balls never appear on a path. (In other words, new balls arriving at a node can only come from nodes higher on the same path.) We show that at least a constant fraction of balls escapes from each path once every two phases. Intuitively, this analysis captures how fast balls disperse within the tree.

Formally, let us fix a phase ϕ and a path π from the root to a parent of a leaf. Let $\eta_1, \eta_2, \ldots, \eta_{\log n}$ be the nodes on π, ordered by depth. A *gateway* node (or simply a gateway) is a child of η_i that is not on π. For uniformity, we combine both children of the


(a) entire tree

(b) 5 balls on the rightmost path and 5 empty bins reachable from the path

Figure 4.3: Closer look at a path in a possible configuration

last node on π (tree leaves) into one gateway meta-child [2]. For instance, in the sample configuration from Figure 4.3a, consider the rightmost path (highlighted in Figure 4.3b); all the left children of nodes on π are gateways. By construction, the sum of remaining capacities of all gateway subtrees (corresponding to empty leaves reachable from π) is equal to the total number of balls on π. In phase ϕ, most balls on the path propose paths going through these gateways.

We now show that, if ball b_i is among the highest priority balls that have chosen the same gateway, then b_i will escape from path π either in phase ϕ or $\phi + 1$.

Lemma 12. *Consider node η_i on π and let c_i be the remaining capacity of its gateway subtree. If m balls choose that subtree in ϕ, then at least* $\min(m, c_i)$ *balls escape π in ϕ and $\phi + 1$.*

Proof Let ball b_k be among the $\min(m, c_i)$ highest priority balls that have chosen the gateway at η_i. Then, b_k is in one of the following scenarios.

Case 1: b_k attempts to move down towards η_i, and stops at some node η_j above η_i. This happens because the subtree down on π has exceeded its capacity. In this case, in $\phi + 1$, b_k tries the gateway subtree at η_j and, by Lemma 4, that subtree has enough space to accommodate b_k.

Case 2: b_k reaches η_i. By assumption, b_k is among $\min(m, c_i)$ highest priority balls that

[2]Obviously, a collision in a subtree with 2 leaves is solved within one phase.

have chosen the same gateway. Thus, b_k escapes π into the gateway subtree of η_i.[3] If b_k is in Case 1, it escapes path π in phase $\phi+1$. If b_k is in Case 2, it escapes π in phase ϕ. There are at least $\min(m, c_i)$ such balls, and the claim follows. □

We now bound the probabilities with which balls try each gateway on π.

Lemma 13. *Let $M_i = \sum_{1 \le j \le i} balls(\eta_j, \phi)$ be the number of balls on the subpath of π from the root to η_i. If c_i is the remaining capacity of the gateway subtree of η_i, then M_i balls try this gateway in ϕ with probability at least c_i/M_i.*

Proof Recall from Lines 5–9 of Algorithm 2 that balls construct paths trying between the children of each node with probabilities indexed by their remaining capacities.

By construction, each ball competes on a subtree with every other ball that can reach the same subtree. Also by construction, the remaining capacity of all subtrees reachable from some path from the root is equal to the number of balls on this path. Note that the total remaining capacity of the subpath from the root to η_i (i.e., all gateway subtrees and the non-gateway subtree at η_i) is M_i. Thus, every ball tries the gateway at η_i with prob. at least c_i/M_i. □

In the following, we show that at least a constant fraction of balls escape π in every two phases.

Lemma 14. *Let M be the total number of balls on π in phase ϕ. For some const. $c > 1$, less than $\frac{M}{c}$ balls have escaped π after $\phi + 1$ with prob. $< e^{\frac{-M}{c}}$.*

Proof By Lemma 13, M_i balls try the gateway at η_i with prob. at least $\frac{c_i}{M_i}$.

By Lemma 12, for any $\gamma > 1$, $\frac{c_i}{\gamma} \le c_i$ highest priority balls that choose the gateway at η_i, escape π in ϕ or $\phi + 1$. Define as success the event that some ball chooses the gateway at η_i. By Lemma 13, such an event occurs with prob. at least c_i/M_i among M_i tries. Thus, the number of successes follows $\mathrm{B}(M_i, \frac{c_i}{M_i})$. From the Chernoff Bound (Fact 3),

[3] In Case 2, ball b_k can stop somewhere else deeper in the tree, but this no longer affects the analysis of π.

$$\Pr\left(\mathrm{B}(M_i, \frac{c_i}{M_i}) < c_i - \frac{c_i}{\gamma}\right) < e^{-\frac{M_i}{\gamma}}.$$

Recall that $\sum_{1 \leq i \leq \log n} c_i = M$. Considering all η_i, $1 \leq i \leq \log n$ on π, the sum of successes is less than $M - \frac{M}{\gamma}$ with prob. $< e^{-\frac{M}{\gamma}}$. Since γ is arbitrary, the claim follows. $\square$

From the above lemma, it follows that all M balls on π escape the path within $O(\log M)$ phases, with probability at least $1 - (1/e)^{\Theta(M)}$.

Lemma 15. *Consider a path π containing M balls. After $O(\log M)$ phases, the probability that π remains non-empty is at most $< e^{-\frac{M}{c'}}$, for some constant $c' > 0$.*

Proof Fix c to be a constant. By Lemma 14, at least $\frac{M}{c}$ balls escape from π after phase $\phi+1$, with probability at least $1-e^{-\frac{M}{c}}$. Starting with M balls, and iterating in Lemma 14 over the remaining balls for $2c \log M$ phases, we obtain that the probability that π is not empty after $2c \log M$ phases is at most $2c \log M e^{-\frac{M}{c'}} < e^{-\frac{M}{(c-\epsilon)}}$ for some small constant $\epsilon < 1$. $\square$

Finally, we combine the two parts of the proof in the following theorem.

Theorem 16. *Balls-into-Leaves terminates in $O(\log \log n)$ rounds with probability at least $\left(1 - \frac{1}{n^c}\right)$, where $c > 0$ is a constant.*

Proof From Lemma 11, after $O(\log \log n)$ phases, the probability there is a path with more than $O(\log^3 n)$ balls, is less than $\frac{1}{n^{c'}}$, for some const. $c' > 0$. Taking $M = O(\log^3 n)$ balls on path π from the root to a parent of some leaf, by Lemma 15 and a union bound over all such $n/2$ paths, the probability that some path is not empty is less than $\frac{1}{n^{c'}}$, for some constant $c' > 0$. Putting together the above results, we get that the probability that the tree still has a populated inner node is at most $\frac{1}{n^{c'+c''}}$ where c', c'' are constants. By construction, the algorithm terminates when all balls have reached leaves. Choosing some constant $c > c' + c''$, the algorithm terminates in $O(\log \log n)$ phases with probability $> 1 - \frac{1}{n^c}$.

Since each phase consists of 2 rounds, the claim follows. $\square$

Message Complexity and Bit Complexity

Since in each round, processes employ all-to-all communication, the total message complexity is $O(n^2 \log\log n)$ messages whp.

In each round processes exchange their ids and up to $\log n$ values in range $\{1, \ldots, 2n\}$. There are $2n$ nodes in the tree, thus each node can be encoded by a value in range $\{1, \ldots, 2n\}$. Thus, the message size is bounded by $O\left(\log n \log N_{max}\right)$ bits, where N_{max} is the largest id.

4.2.3 The Impact of Crashes on Round Complexity

To show that crashes do not slow down termination, we continue the analysis of some path π that starts at the root. By construction, at the end of each phase, in every local view, the position of each surviving ball b_i is updated according to b_i's local view. We thus focus on the progress of b_i in its local view.

Iterating on each phase ϕ which contains at least one failure, we compare the local view V of ball b_i in ϕ with its hypothetical view V' in an execution with a failure free ϕ. First, note that views V and V' are equivalent if b_i has not seen a failure. Now, assume b_i has seen a failure in V.

We show that b_i is at least as likely to escape from π in V, as it is in V'. First, we note that if b_i has seen a failure in some disjoint subtree, or the crashed ball had lower priority than b_i, then, by construction, such a failure does not affect the progress of b_i. Consider now a failure which occurs in a subtree of b_i. Such a failure implies that, in V, the total capacity of all gateways on π is larger than the number of balls on π. Thus, b_i is at most as likely to be among the first highest priority balls that have chosen the same gateway in ϕ. Since the choice of b_i and π is arbitrary, the argument applies to every ball in every view.

4.2.4 Experimental Evaluation

We complement with simulations the analytical analysis of the round complexity of Balls-into-Leaves. The results are depicted in Table 4.1 (the number of rounds was obtained by averaging over 200 failure-free runs of the algorithm).

Number of balls	Number of rounds
2^5	2.76
2^{10}	4.00
2^{15}	4.92
2^{20}	5.24
2^{25}	6.00

Table 4.1: Experimental evaluation of Balls-into-Leaves

Our simulations confirm that the running time grows very slowly with n: for example, for 2^{25} balls the algorithm takes only 6 rounds to terminate. These results are consistent with our $O(\log \log n)$ bound. Moreover, the simulations show that the hidden constants in our asymptotic analysis are indeed are small.

4.3 Early Terminating Extension

We now extend the algorithm to terminate faster in executions with fewer crashes. Without failures, balls can use their unique labels to pick distinct leaves in one round: balls exchange their labels, and each ball chooses a leaf indexed by the rank of its label in the ordered set of all labels. But collisions may occur due to failures. A single crash can cause up to $n/2$ collisions: the ball with the lowest label b_{lowest} sends to every second ball (by label order) and then crashes, so that all other balls collide in pairs.

On the other hand, if balls use this scheme to deterministically pick paths in the Balls-into-Leaves algorithm, it is easy to see that the paths are well distributed; in the first phase, the tree collapses into small subtrees of depth 2 in every local tree. But the balls cannot use deterministic paths in every phase, otherwise the algorithm's round complexity would be no better than $\Omega(\log n)$, due to the lower bound of [CHT99].

We combine the deterministic and randomized approaches, by first deterministically collapsing the tree into disjoint subtrees of depth $O(\log f)$ (where f is a number of failures that have occurred in an execution), and then resorting to randomization. The modified Balls-into-Leaves algorithm works as follows. In Round 1 of phase 1, replace Lines 6–11 in Algorithm 2 with the following: ball b_i constructs path deterministically towards the leaf ranked by b_i in OrderedBalls(); the rest of phase 1 is executed as in the original algorithm. In the remaining phases, b_i executes the code of the original algorithm.

It is easy to see that, in a failure-free execution, the modified algorithm terminates in one phase.

Theorem 17. *In a failure-free execution, the modified Balls-into-Leaves algorithm terminates in $O(1)$ rounds.*

In an execution with f failures, we show that the algorithm terminates in $O(\log\log f)$ rounds whp.

Theorem 18. *The modified Balls-into-Leaves algorithm terminates in $O(\log\log f)$ rounds with prob. at least $\left(1 - \frac{1}{n^c}\right)$, where $c > 0$ is a constant.*

Proof Let i be rank of ball b_i among the surviving balls. Assume b_i has not seen $k \leq f$ failures. The rank in its local view has shifted right by at most k with regard to other views. On the other hand, all surviving balls have b_i in their local views. Thus, at most $k-1$ other survivors see their ranks in the interval $i..(i+k)$.

Consider binary representation of leaf ranks. We note that each subtree that contains some leaf is indexed by the binary prefix of the leaf rank. From the previous discussion, the surviving balls collide on at most $\lceil \log f \rceil$ least significant bits. Thus, in every local view, collisions occur at the depth at least $\log n - \lceil \log f \rceil$ in phase 1.

In the subsequent phases, balls in disjoint subtrees propose non-overlapping paths. Therefore, the rest of the execution is equivalent to running at most $\frac{n}{2^{\log n - \lceil \log f \rceil}} \leq n$

parallel instances of Balls into Leaves with at most f balls each. From Theorem 16, and for a sufficiently large const. $c > 0$, the claim follows. $\square$

Chapter 5

Renaming with Byzantine Faults

In this chapter, we address renaming in the context of more challenging Byzantine faults and assume that up to $t < n$ processes may be Byzantine. We first consider the weaker non-rushing adversary that sends messages at the beginning of each round without having access to random choices of correct processes. In this setting, due to a possibly high number of corrupted processes, it is no longer possible to resort to techiques that require close coordination among processes (as in the previous chapter). We propose an algorithm based on the following simple idea: each process randomly chooses a new name from the range $\{1, \ldots, n\}$ and applies tie-breaking rules to choose a winner in the case of a collision. Each process repeats the procedure until it wins a name. We show that this algorithm terminates in $O(\log n)$ rounds whp.

We next consider a stronger *rushing* adversary that sees the random choices of correct processes at the beginning of each round. We show that our first algorithm can tolerate $t = 1$ Byzantine failure but the algorithm fails if $t > 1$. This is because, in each round, Byzantine processes can first observe the choice of a correct process and mimic it, causing infinitely many collisions. To tackle this problem, we use a cryptographic commitment primitive to force processes to commit to a choice without revealing the value; this technique prevents Byzantine processes from constantly mimicking the choices of correct processes. We extend the previous algorithm with cryptographic commitment;

the extended algorithm works for any $t < n$. To use cryptographic commitment, we must assume a polynomially bounded adversary, so that the adversary cannot break the cryptographic primitive.

5.1 Renaming with the Non-Rushing Adversary

In this section, we consider the non-rushing adversary and present an algorithm (Algorithm 3) that solves renaming in $O(\log n)$ rounds. The algorithm solves tight renaming, where $M = n$.

The algorithm is based on the following idea. A process chooses uniformly at random a name from $\{1, \ldots, n\}$ and exchanges its choice with all other processes. If no other process picks the same name, the process is done and informs all other processes. Otherwise, the process excludes names already chosen by other processes and then restarts. Since the adversary cannot inspect the random choices of the correct processes before issuing its messages, each process will eventually pick a unique name. All Byzantine processes can do is to repeatedly claim t distinct available names to increase the chance of collisions. For example, in the presence of $n-1$ Byzantine processes, the correct process has a chance of $1/n$ of not colliding with the choices of the Byzantine processes. Hence, without additional rules, the expected decision time for a correct process could be linear in the size of the network.

To speed up the decision time, we use a tie breaking rule that allows processes to decide even in the case of collisions. Tie breaking is done by appointing in each phase a set of processes whose choices, in case of a collision, have priority over the choice of the given process. These sets are calculated in each phase as follows. In odd phases, the set consists of the undecided neighbors with ids lower than *myId* (the id of the current process); in even phases, the set consists of undecided neighbors with ids higher than *myId*. Roughly speaking, with this rule, in a given phase, a correct process competes with some fraction of undecided processes, and, in the following phase, competes with the

Algorithm 3 Renaming with $t < n$

```
01 Initialization:
02     $undecided := \{1, \ldots, n\}$;          // set of links to undecided neighbors
03     $freenames := \{1, \ldots, n\}$;          // set of available names
04     foreach $i \in \{1, \ldots, n\}$ do
05         $ids[i] := \perp$;                    //array that stores old ids of all neighbors

06 In Phase 1 do
       // Phase 1 has a single round
07     send $\langle \mathrm{ID}, myId \rangle$ to all links;
08     foreach $i \in \{1, \ldots, n\}$ do
09         if $\langle \mathrm{ID}, id \rangle$ has been received from link $i$ then
10             $ids[i] := id$;
11     proceed to Phase 2;

12 In Phase $\phi > 1$ do
       // Round 1
       // choose priority set for the current phase
13     if $\phi$ is odd then
14         $plinks := \{i \in undecided : ids[i] < myId\}$;
15     else
16         $plinks := \{i \in undecided : ids[i] > myId\}$;
17     $myName$ := choose an element in $freenames$ uniformly at random;
18     send $\langle \textsc{proposal}, myName \rangle$ to every link $i$ such that $i \in undecided \setminus plinks$;
       // check for collisions
19     if $\exists i \in plinks : \langle \textsc{proposal}, name_i \rangle$ has been received from link $i$
       such that $myName = name_i$ then
20         $winner := false$;
21     else
22         $winner := true$;

       //Round 2
23     if $winner = true$ then
24         send $\langle \textsc{decided}, myName \rangle$ to every link $i$ such that $i \in undecided$;
25         return $myName$;
26     else
27         foreach $i \in undecided$ do
28             if $\langle \textsc{decided}, name_i \rangle$ has been received from link $i$ then
29                 $undecided := undecided \setminus \{i\}$;
30                 $freenames := freenames \setminus \{name_i\}$;
31         proceed to Phase $\phi + 1$;
```

remaining fraction of undecided processes. As we will show, this tie breaking rule reduces the decision time to a factor of $\log n$. Hence, the algorithm terminates in $O(\log n)$ rounds w.h.p (i.e., with probability at least $\left(1 - \frac{1}{n^c}\right)$, where $c > 0$ is a constant).

We now describe the algorithm in detail. Each process p_i stores the following data structures:

- ids_i: an array that stores ids (not names) that p_i knows about, indexed by the link to which they are connected, i.e. $ids_i[j]$ stores the id of the neighbor connected to link j of p_i. Initially, $ids_i[j] = \perp$ for every j.
- $freenames_i$: the set of names that p_i believes to be free (not selected by its neighbors). Initially, $freenames_i = \{1, \ldots, n\}$ (all possible names).
- $undecided_i$: the set of links to neighbors that p_i believes to not have decided yet. Initially, $undecided_i = \{1, \ldots, n\}$ (all links).

The purpose of the first phase of the algorithm is for processes to exchange their initial ids. Each process sends its own id to all links and stores the ids it receives from other links (Lines 07–10). The ids received in phase 1 are used in all subsequent phases.

The following phases are tournament phases, where processes compete for available names. There are two ways of winning a tournament: if there are no collisions on the name chosen by a process in the current phase (i.e., the process was the only one to make that choice) or, when collisions occur, by winning a tie breaking rule, which differs according to the phase number. The tie breaking rule updates *plinks* in each phase as follows. In odd phases, *plinks* includes undecided neighbors with ids lower than *myId*; in even phases, *plinks* includes undecided neighbors with ids higher than *myId* (Lines 13–16). In what follows, recall that $link_j(p_i)$ denotes the number at process p_j of the bidirectional link to p_i (recall from Chapter 3). The relevant property of the *plinks* assignment is that, for any phase ϕ, when considering any two correct processes p_i and p_j, if $link_i(p_j) \in plinks_i$, then $link_j(p_i) \notin plinks_j$. Furthermore, if $link_i(p_j) \in plinks_i$ in ϕ, and p_i and p_j do not decide in ϕ, then $link_j(p_i) \in plinks_j$ in $\phi + 1$.

Each tournament phase has two rounds. In the first round, a correct process selects *plinks*, picks randomly a name from *freenames*, and sends the name to the undecided neighbors that do not belong to *plinks* (Lines 17–18). There is no need to send the name to the neighbors in *plinks* because if a collision occurs in this phase, they will ignore the choice of the given process. A process wins the tournament if it has not received the same name from any process in *plinks*. At the end of the first round, processes check if they have won the tournament (Lines 19–22).

In the second round, if a correct process has won the tournament, it sends a $\langle \textsc{decided}, myName \rangle$ message to all undecided neighbors and terminates, returning variable *myName* as its new name (Lines 23–25). Otherwise, if the process did not win, it collects $\langle \textsc{decided}, name_i \rangle$ messages from its neighbors, removes these neighbors from *undecided* (doing so excludes these neighbors from all succeeding tournaments), and removes the names elected by the decided processes from *freenames* (Lines 26–30).

Each undecided process keeps executing consecutive tournament phases until it wins a name in one of the tournaments.

Analysis

In the following we prove the correctness of Algorithm 3. Whenever needed to distinguish between local variables at distinct processes, we use subscript $_i$ to indicate the local variables of process p_i. Superscript $^\phi$ indicates the value of a variable in Round 1 of phase ϕ.

Lemma 19. *For any phase $\phi > 0$ and any correct processes p_i and p_j, if p_i has not decided a new name before ϕ, then $link_j(p_i) \in undecided_j$ in Round 1 of ϕ.*

Proof Assume there exist correct undecided processes p_i and p_j such that $link_j(p_i) \notin undecided_j$ in Round 1 of ϕ. By the algorithm, the links to all neighbors are initially in $undecided_j$ (Line 02) and are excluded from the set only when a $\langle \textsc{decided}, \cdot \rangle$ message is received from the corresponding neighbor (Lines 28–29). Thus, p_j must have previously received $\langle \text{DECIDED}, v_i \rangle$ from p_i. This, in turn, means that p_i sent $\langle \text{DECIDED}, v_i \rangle$ before ϕ (Lines 24–25). But by assumption p_i is undecided in Round 1 of ϕ—a contradiction. □

The following lemma states that each correct process considers available at least as many names as there are processes participating in each tournament phase.

Lemma 20. *For every correct process p_i, in Round 1 of any phase $\phi \geq 2$,*

$$|\mathit{undecided}_i| \leq |\mathit{freenames}_i| \, .$$

Proof The proof follows by induction on the phase number.

Base case. By the algorithm, p_i starts with $|\mathit{undecided}_i| = |\mathit{freenames}_i| = n$ (Lines 02–03). Thus, in Round 1 of phase $\phi = 2$, $|\mathit{undecided}_i| = |\mathit{freenames}_i|$.

Induction step. Assume $|\mathit{undecided}_i| \leq |\mathit{freenames}_i|$ in Round 1 of phase ϕ. In Round 2 of the same phase, p_i only accepts $\langle \text{DECIDED}, . \rangle$ messages from the links in $\mathit{undecided}_i$ (Line 27-28). For each link $j \in \mathit{undecided}_i$, if $\langle \text{DECIDED}, v_j \rangle$ message has been received from j in the current phase, p_i removes j from $\mathit{undecided}_i$; and, if $v_j \in \mathit{freenames}_i$, p_i also removes v_j from $\mathit{freenames}_i$ (Lines 28–30). As a result, in Round 1 of phase $\phi + 1$, $|\mathit{undecided}_i| \leq |\mathit{freenames}_i|$. □

The following lemma establishes the *uniqueness* property of the algorithm based on two following observations: in case of a collision, at most one correct process (with the smallest id in odd phases and with the largest id in even phases) wins the tie breaking; the winning process always announces its decision before returning.

Lemma 21. *No two correct processes decide on the same name in Algorithm 3.*

Proof Assume, by contradiction, that there are two correct processes p_i and p_j that decide on the same name v in phases ϕ_i and ϕ_j respectively. We will distinguish two possible scenarios.

Case 1. $\phi_i \neq \phi_j$. Without loss of generality, assume $\phi_i < \phi_j$. By the algorithm, if p_i decided on v in ϕ_i, then p_i sent $\langle \textsc{decided}, v \rangle$ in Round 2 of ϕ_i to all undecided neighbors before returning the new name (Lines 24–25). By Lemma 19, $\mathit{link}_j(p_i) \in \mathit{undecided}_j$ and $\mathit{link}_i(p_j) \in \mathit{undecided}_i$ in Round 1 of ϕ_i. Therefore, p_i sent v to p_j, and p_j excluded v from $\mathit{freenames}_j$ in Round 2 of ϕ_i (Lines 27–30). On the other hand, if p_j decided on v in phase ϕ_j, then p_j must have chosen v from $\mathit{freenames}_j$ in ϕ_j (Line 17), which contradicts the previous statement.

Case 2. $\phi_i = \phi_j = \phi$. Without loss of generality, assume that $id_i < id_j$ and ϕ is odd (for even values of ϕ the argument is symmetric). If both p_i and p_j decided on v in Round 2 of ϕ, then p_i and p_j were both undecided in Round 1 of ϕ.

Also, by Lemma 19, $\mathit{link}_j(p_i) \in \mathit{undecided}_j$ and $\mathit{link}_i(p_j) \in \mathit{undecided}_i$ in Round 1 of ϕ. Processes p_i and p_j randomly picked v from *freenames* and sent v to their neighbors in $\mathit{undecided} \setminus \mathit{plinks}$ (Lines 17–18). By assumption, $id_i < id_j$; therefore, $\mathit{link}_j(p_i) \in \mathit{undecided}_j \setminus \mathit{plinks}_j$. Hence, p_i sent v to p_j in Round 2 (Line 18). Since $\mathit{link}_j(p_i) \in \mathit{plinks}_j$, p_j received v from p_i (Lines 19–20). But by assumption, p_j decided on v in Round 2, which is possible only if p_j had not received v from any link in plinks_j in Round 1 (Lines 19–22)—a contradiction. □

Round Complexity

In the following we calculate the round complexity of the algorithm. Recall that we consider the *non-rushing* adversary, whose behavior is independent from the random choices of the correct processes in the current round.

We now prove that each correct process decides whp after $O(\log n)$ rounds. To do so, we consider an arbitrary correct process p_0 and give an upper bound on the probability that p_0 has not decided after $O(\log n)$ rounds.

Intuitively, the adversary can decrease the probability that p_0 decides in some phase by making certain Byzantine processes (those not in plinks_0) announce their decision in the previous phase. However, once a process decides, it is excluded from subsequent tournaments, and so the adversary can do this only a limited number of times. We will show that the adversary has to make a large fraction of processes to decide in order to decrease by a constant factor the probability that p_0 does not decide. Thus, in each phase, the adversary has to carefully balance the number of processes that it causes to decide and the probability that p_0 does not decide. We will show that, whatever the strategy of the adversary, after $O(\log n)$ rounds, p_0 decides whp.

Lemma 22. *Under the non-rushing adversary, the probability that a correct process p_0 decides by round $12 \log n + 3$ is at least $1 - \frac{1}{n^2}$.*

Proof In the analysis, we assume without loss of generality that the adversary controls *all* processes other than p_0; if the probability lower bound holds in this case, then it also holds if the adversary can control a smaller number of processes. We say that a link ℓ decides at an epoch e (or at a round r) if p_0 receives a $\langle \mathit{decided}, \cdot \rangle$ message from link ℓ in epoch e (or round r). Intuitively, we are considering a p_0-centric view of the system since p_0 is the only correct process. We say that *link ℓ is undecided at epoch e (or round r)* if ℓ has not decided at epoch e (round r) or earlier.

Recall that the algorithm is organized in phases of two rounds each. We will further group two consecutive phases into an epoch $e \geq 1$: epoch e has phases $2e + 1$ and $2e + 2$. For convenience we also define epoch 0 as having just a single phase, phase 2.

In each epoch, we pick one phase to scrutinize more carefully; this is called the *chosen phase* of the epoch. For the other phase, we will use a trivial upper bound of 1 on the probability of non-termination. The chosen phase of epoch e is determined as follows. At the beginning of last round before epoch e,[1] consider the sets $pl1 = \{i \in \mathit{undecided}_0 : \mathit{ids}_0[i] < \mathit{myId}_0\}$; and $pl2 = \{i \in \mathit{undecided}_0 : \mathit{ids}_0[i] > \mathit{myId}_0\}$, where $\mathit{undecided}_0$, $\mathit{ids}_0[i]$, and myId_0 are variables of process p_0. If $|pl1| < |pl2|$ then we pick the first phase of epoch e as the chosen phase, otherwise we pick the second phase. Intuitively, we want the phase with fewest priority links, where $pl1$ is the set of priority links of the first phase and $pl2$ is the set of priority links of the second phase (see Lines 13–16 of Algorithm

[1]The last round before epoch e is when the adversary can make links decide before starting epoch e.

3). We let pl_e and npl_e be the priority links and non-priority links, respectively, of the chosen phase evaluated at the beginning of last round before epoch e. That is, $pl_e = pl1$, $npl_e = pl2$ if the chosen phase is the first phase, and $pl_e = pl2$, $npl_e = pl1$ if the chosen phase is the second phase.

For each epoch e, we define the following variables:

- n_e= the number of undecided links at the beginning of the last round before epoch e;
- $\gamma_e = |pl_e| / n_e$;
- $\alpha_e = \frac{|links\ in\ npl_e\ that\ decide\ before\ the\ chosen\ phase\ of\ e|}{n_e}$;
- $\bar{\alpha}_e = 1 - \alpha_e$;
- c_e = number of links in pl_e that decide before the chosen phase of epoch e.

We now bound the probability P_e that p_0 does not decide in the chosen phase of epoch e, as follows:

$$P_e \leq \frac{n_e\gamma_e - c_e}{n_e\bar{\alpha}_e - c_e} \leq \frac{n_e\gamma_e}{n_e\bar{\alpha}_e} \leq \frac{1}{2\bar{\alpha}_e} \tag{5.1}$$

Here, the first inequality holds because the best strategy for the adversary to delay termination is to claim as many free names as it can (at most $n_e\gamma_e - c_e$, the number of priority links) over the total number of free names (at least $n_e\bar{\alpha}_e - c_e$, the number of undecided links). The second inequality holds because $(a-c)/(b-c) < a/b$ for any positive integers a, b, c such that $a < b$ and $c < b$. The third inequality holds because $\gamma_e \leq 1/2$ by definition of γ_e and of the chosen phase.

We can trivially upper bound by 1 the probability that p_0 does not decide in the non-chosen phase of epoch e. Thus, from (5.1), the probability that p_0 does not decide in epoch e is upper bounded by $\frac{1}{2\bar{\alpha}_e}$ as well. Therefore, the probability $\mathcal{P}_e$ that p_0 does not decide in any of epochs $1, \dots, e$ is upper bounded by

$$\mathcal{P}_e \leq P_1 \times \cdots \times P_e \leq \frac{1}{2^e \bar{\alpha}_1 \dots \bar{\alpha}_e} \tag{5.2}$$

Note that $n_{e+1} \leq n_e \bar{\alpha}_e$ because at least $\alpha_e n_e$ links decide before the chosen phase of epoch $e+1$. Therefore,

$$n_{e+1} \leq n_1 \bar{\alpha}_1 \bar{\alpha}_2 \dots \bar{\alpha}_e \tag{5.3}$$

We do not know the values of $\bar{\alpha}_i$ because they depend on the strategy of the adversary. However, from (5.2) and (5.3), we can upper bound the product $\mathcal{P}_e n_{e+1}$:

$$\mathcal{P}_e n_{e+1} \leq n_1 / 2^e \leq n / 2^e \tag{5.4}$$

For $e = 3 \log n$, we obtain $\mathcal{P}_e n_{e+1} \leq 1/n^2$. If p_0 has not decided by the end of epoch e then $1 \leq n_{e+1}$. Therefore, $\mathcal{P}_e \leq \mathcal{P}_e n_{e+1}$, and so $\mathcal{P}_e \leq 1/n^2$. Note that each epoch has two phases, and each phase has two rounds. Moreover, there are three initial rounds before epoch 1. Thus, there are $3 + 12 \log n$ rounds until the end of epoch e. Therefore, the probability that process p_0 has not decided by round $3 + 12 \log n$ is upper bounded by $1/n^2$. □

We now use Lemma 22 to calculate the overall round complexity of Algorithm 3.

Theorem 23. *Under the non-rushing adversary, all correct processes decide on names in $O(\log n)$ communication rounds whp.*

Proof By Lemma 22, the probability that a correct process decides in at most $12 \log n + 3$ rounds is at least $1 - \frac{1}{n^2}$. By taking a union bound, the probability that there exists at least one correct process that has not decided after $12 \log n + 3$ rounds is at most $\frac{1}{n}$. □

We now have all ingredients necessary to prove the correctness of Algorithm 3.

Theorem 24 (Correctness)**.** *Under the non-rushing adversary, Algorithm 3 solves tight renaming for any $t < n$.*

Proof

Validity condition is satisfied by the algorithmic construction: processes propose new names from the set $\{1, \ldots, n\}$ and decide only on the values they have proposed.

Uniqueness follows from Lemma 21.

Termination with probability 1 follows from Theorem 23. □

Message Complexity and Bit Complexity

Since in each round, processes employ all-to-all communication, the total message complexity is $O(n^2 \log n)$ messages whp.

In the first round, processes send their ids; hence the message size is bounded by $O(\log N_{max})$ bits, where N_{max} is the largest id. In the following rounds processes exchange values in range $\{1, \ldots, n\}$; hence the message size in these rounds is bounded by $O(\log n)$ bits.

5.2 Renaming with the Rushing Adversary

In this section, we consider the rushing adversary, which is allowed to inspect the messages from correct processes *before* Byzantine processes send their own messages. As a result, in our algorithm, the Byzantine processes can echo the choices of each correct process causing infinitely many collisions. Surprisingly, even in this case, correct processes are still able to decide on new names in the presence of *one* Byzantine process.

5.2.1 Case of $t = 1$

We now show that the algorithm of Section 5.1 works under the rushing adversary for $t = 1$. The *validity* property follows from the algorithm construction. *Uniqueness* follows from Lemma 21.

It remains to show the *termination with probability 1.* Consider an execution of Algorithm 3 with $t = 1$. For any $\phi \geq 2$, if a correct process has a link to the Byzantine process in the *plinks* set of phase ϕ, then this process will ignore the choice of the Byzantine process in $\phi + 1$. As a result, the Byzantine process can influence the outcome of a half of tournament phases, while in the remaining phases the correct process is competing only with the undecided correct neighbors.

Theorem 25. *Under the rushing adversary with $t = 1$, Algorithm 3 terminates in $O(\log n)$ rounds whp.*

Proof Let p_i be an undecided correct process. For any two consecutive phases there exists a phase, say ϕ, when the link to a Byzantine process $\notin plinks_i^{\phi}$. Therefore, the choice of the Byzantine process is ignored by p_i in every such ϕ (Line 19). Since by Lemma 20, in each phase there are as many free names as undecided processes, the expected decision time for p_i is at most double the decision time under the non-rushing adversary (see Lemma 22 and Theorem 23). □

5.2.2 General Case of $t > 1$

If $t > 1$, two Byzantine processes can announce to a correct process p_i a smaller id and a larger id than p_i's identifier, and then generate collisions deterministically in each tournament phase. In this way, the correct processes are never able to decide. To prevent such behavior, we introduce in Algorithm 3 a cryptographic commitment primitive. In the modified algorithm, depicted in Algorithm 4, the random choices of processes are not announced immediately. Instead, the undecided processes first commit to their choices without revealing the actual values, and in a subsequent round reveal their choices. Thus, the adversary is required to commit the values of the corrupted processes without knowing the values committed by the correct processes (hiding property). Furthermore, the adversary is not able to modify its choices during the revealing stage (binding property).

Algorithm 4 Renaming with the Use of Commitment

```
// In Algorithm 3 replace the lines
17      send ⟨PROPOSAL, $myName$⟩ to every link $i$ such that $i \in undecided \setminus plinks$;
18      if $\exists i \in plinks$ : ⟨PROPOSAL, $name_i$⟩ has been received from link $i$
        such that $myName = name_i$ then

// by the following
17      COMMIT ($\phi$, $myName$, $myId$) to every link $i$ such that $i \in undecided \setminus plinks$;

        // Round 2
18a     REVEAL ($\phi$, $myName$, $myId$) to every link $i$ such that $i \in undecided \setminus plinks$;
18b     if $\exists i \in plinks$ : such that ($\phi, name_i, ids[i]$) has been revealed from link $i$
            and $myName = name_i$ then
```

Therefore, the algorithm operates analogously to its non-rushing counterpart.

As noted before in the text, with small probability, the adversary is able to break the commitment. In this case, it will be able to reproduce the value of a correct process, causing a collision. However, since in different phases processes use independent instances of commitment, correct processes still decide with probability 1. Therefore, under the computationally bounded adversary, with probability 1, Algorithm 4 terminates correctly.

Commitment abstraction can be implemented in constant number of rounds, e.g. [PR05]. Hence, the total round complexity of Algorithm 4 is logarithmic.

Theorem 26. *Under the rushing computationally bounded adversary, the modified algorithm solves renaming in $O(\log n)$ rounds whp.*

Chapter 6

Order-Preserving Renaming with Byzantine Faults

In this chapter, we continue with the Byzantine fault model and turn our attention to a stronger order-preserving variant of renaming. Recall that in order-preserving renaming, processes' new names are required to preserve the order of their original ids. To our knowledge, we are the first to address this problem in the Byzantine model.

We show that order-preserving renaming can be solved efficiently in this model, but only if the number of Byzantine faults is bounded by $t < n/3$. In particular, we show that order-preserving renaming can be solved in $O(\log n)$ rounds and with target namespace of size $n + t - 1$, if $t < n/3$. In this algorithm, processes exchange their ids, sort all received ids, and propose a new name for each id based on its rank in the ordered set. Then, processes run a variant of approximate agreement on new names for all ids to ensure that the names are in the correct order. After presenting this algorithm and the proofs, we proceed by showing that if t is bounded by $O(\sqrt{n})$, the algorithm solves order-preserving renaming in $O(1)$ rounds and with *tight* namespace. We then present a simple double-echo algorithm for small values of t that solves order-preserving renaming in just 2 rounds.

On the negative side, we show that order-preserving renaming cannot be solved if $t \geq n/3$; the impossibility applies to both deterministic and randomized algorithms. The proof is by the indistinguishability argument: we assume by contradiction the existence of an algorithm that solves order-preserving renaming with $t \geq n/3$ and construct indistinguishable executions, in which a correct process decides on different values. This impossibility result implies a separation between the power of renaming and order-preserving renaming.

6.1 Algorithm for $n > 3t$

In this section, we present what is, to our knowledge, the first order-preserving renaming algorithm tolerant to Byzantine faults. The resiliency of our algorithm is $n > 3t$. At a high level, our algorithm follows the structure of the order-preserving algorithm for the crash fault model [Oku10], employing the techniques of Byzantine approximate agreement (AA) [DLP+86] with extensions that address the two following additional concerns. First, when Byzantine processes announce their ids, they can send different values to different correct processes. Therefore, when the computation requires a list of existing ids, processes may include some faulty ids, which other correct processes may regard as possibly correct (note that it would be too costly to agree on a unique set of ids). The first challenge is to limit the number of such faulty ids. Second, we need to ensure that, in spite of contradictory information sent by corrupted processes, the instances of AA converge in a way that preserves the initial ordering.

The algorithm, depicted in Algorithm 5, uses two distinct phases, namely the id selection phase and the rank approximation phase, or *voting*. The first phase takes 3 rounds and aims at limiting the number of ids announced by Byzantines processes that are not recognized as faulty, while ensuring that all correct processes know all correct ids. At the end of this phase, each process makes an estimate of the new name for each process. Since these estimates are not precise enough to be order preserving, the second phase of the algorithm runs, in parallel, coordinated Byzantine-tolerant approximate agreements on those estimates. This phase is called approximation phase and takes logarithmic number of rounds. We denote each round of the approximation phase as a *voting round*. By making appropriate validations on the votes of each process, we ensure that the values converge preserving the order of original ids. Below we discuss each of these two phases in detail.

6.1.1 Id Selection Phase

The id selection phase is implemented in Rounds 1 to 3 of Algorithm 5. The purpose of this phase is to choose which identifiers should feed the rank approximation phase. Note that Byzantine processes can announce different ids to different peers; if their power is not constrained the number of "fake" ids may prevent correct processes from executing correctly. On the other end, we do not aim at ensuring that all correct processes select the exact same set of identifiers: that would be equivalent to solving consensus, which would have linear round complexity. For convenience of exposition, ids belonging to correct processes are named *correct* ids. All other ids are referred to as *Byzantine*, e.g. ids issued by Byzantine processes as their own or non-existent ids that Byzantine processes claim to have received from others.

Each process locally stores the following variables: two sets, *timely* and *accepted*, that are used to collect ids; variable *ranks*, a sparse array where *ranks*[*id*] stores a new name for each id in the *accepted* set. Function SORT(*set*) orders the entries in a set *set*; function RANK(*set*, *v*) returns a position of value v in the ordered set *set*.

At the end of the id selection phase, the following properties are ensured on the *timely* and *accepted* sets:

- at every correct process p, timely_p includes *all* correct ids;
- at every correct process p, $\mathit{accepted}_p$ includes at most $n + t - 1$ ids in total;
- at every correct process p, $\mathit{accepted}_p$ is such that:

$$\bigcup_{q\,:\,q\ is\ correct} \text{timely}_q \subseteq \text{accepted}_p,$$

i.e., if one id is considered timely by some correct process, this id is for sure included in

Algorithm 5 Order-preserving Byzantine Renaming

```
Init:
    $\delta = 1 + \frac{1}{3(n+t)}$; Ids := $\emptyset$; timely := $\emptyset$; accepted := $\emptyset$;

// id selection phase

In Round $r$ := 1
    broadcast ($\langle$ID, $my\_id\rangle$);
    foreach $id$: $\langle$Id, $id\rangle$ received from a distinct link do
        Ids := Ids $\cup$ $\{id\}$;

In Round $r$ := 2
    foreach $id \in$ Ids do
        broadcast($\langle$Echo, $id\rangle$);
    foreach $id$: $\langle$Echo,id$\rangle$ received from at least $n - t$ distinct links do
        Ids := Ids $\cup$ $\{id\}$;

In Round $r$ := 3
    foreach $id \in$ Ids do
        broadcast($\langle$ Ready, $id\rangle$);
    foreach $id$: $\langle$Ready, $id\rangle$ received from at least $n - t$ distinct links do
        timely := timely $\cup$ $\{id\}$;
    foreach $id$: $\langle$Ready, $id\rangle$ received from at least $n - 2t$ distinct links do
        accepted := accepted $\cup$ $\{id\}$;
    sort (accepted);
    foreach $id \in$ accepted do
        ranks[$id$] := rank(accepted,$id$)$\times\delta$;

// rank approximation phase

In Round $r$ := 4 to $2\lceil \log t \rceil + 8$
    votes := $\emptyset$;
    broadcast ($\langle$AA, ranks$\rangle$);
    foreach $\langle$AA, R$\rangle$ received do
        if isValid (timely, R) then
            votes := votes $\cup$ R;
    ranks := approximate(ranks, votes); // updates "accepted" set

    if Round $r = 2\lceil \log t \rceil + 8$
        return round(ranks[$my\_id$]);
```

the *accepted* set by every other correct process (but not necessarily considered timely).

In detail, the first phase of the algorithm works as follows. In Round 1, each correct process broadcasts its identifier in an Id message. In Round 2, processes echo the ids they have received in the previous round (Echo messages). Only ids that have been echoed at least $n-t$ times are considered for the following round. This effectively limits the number of Byzantine ids. Also, since all correct ids are echoed by the correct processes, all correct ids are taken to the next round. Ids that satisfy the previous condition are broadcast in a Ready message in Round 3; all ids for which at least $n-t$ Ready messages have been issued are added to the *timely* set. Notice that all correct ids will be included in the *timely* set of every correct process. All ids for which at least $n-2t$ Ready messages have been produced are added to the *accepted* set. As a result, *accepted* contains all ids in the *timely* set.

The concept of separating the ids in *timely* and *accepted* sets is similar to grading the delivered messages with confidence levels, as done in Gradecast. The classical broadcast [BT85] and Gradecast [FM88] algorithms require each process to know the identity of a sender. Therefore, if the ids are not known *a priori* and all processes are broadcasting at the same time, Byzantine participants can collude such that more than t messages issued by Byzantine processes are delivered by the correct processes. In fact, any message received in the first round by at least $n-2t$ correct processes can be delivered by a correct process. Therefore, in our id selection, the size of the *accepted* set at a correct process can contain as many as $n+t-1$ ids. Note also that Byzantine processes may use correct ids as their own; this has no effect on the execution: duplicate identifiers do not appear in *timely* and *accepted* sets.

At the end of the id selection phase, each process sorts its *accepted* set, and estimates a new name to each of these ids (including its own), which is the rank of that id in the sorted set stretched by factor $\delta = 1 + \frac{1}{3(n+t)}$. This factor is large enough to prevent names from clashing due to small disagreement errors in the approximate agreement, as we explain below. The purpose of the second phase is to iteratively execute approximate

Algorithm 6 Procedure isValid

```
Function isValid (timely, ranks) returns boolean is
    foreach $id$, $id'$ ∈ timely such that $id < id'$ do
        if $id \notin$ ranks or $id' \notin$ ranks or ranks[$id'$]−ranks[$id$]$< \delta$ then
            return false;
    return true;
```

agreement until the ranks calculated by the correct processes are within safe distance.

6.1.2 Approximation Phase

The approximation phase, or *voting*, starts in Round 4 and takes a logarithmic number of rounds to converge. This phase is based on the Byzantine-tolerant AA algorithm of [DLP+86]. The AA algorithm guarantees that, in spite of contradictory inputs from Byzantine processes, the output values are within a bounded error. Moreover, it guarantees that the outputs are within the range of input values issued by the correct processes. In our case, the ranks calculated at the end of the id selection phase may not preserve the correct global ordering. As a result, the ranges of the correct inputs into AA may overlap. Without any additional care, AA may converge to values that are not order preserving.

The above issue is addressed by the verification function depicted in Algorithm 6 that aims at ensuring that the approximation is performed in accordance with the ordering of the original ids. The function isValid takes as input the *timely* set of a local process and array *ranks* received from some other process. It makes two tests to check if the votes from the remote process are consistent. First, the votes must include a vote for each id in *timely* (we remind that if p and q are correct, then $timely_p \subseteq accepted_q$, thus any vote that does not satisfy this invariant may be discarded as faulty). Second, it ensures that the new rankings for these ids appear in the correct order separated by the minimum safety margin of δ. As a result, even if a Byzantine process sends different votes to different processes and both are considered valid, the presented validity conditions are sufficient

Algorithm 7 Procedure APPROXIMATE

```
Function APPROXIMATE (my_ranks, all_ranks) returns array of ranks is
    new_ranks := $\emptyset$;

    foreach $id \in$ accepted do
        votes[$id$] := $\emptyset$;
        foreach R $\in$ all_ranks do
            if $id \in$ R then
                votes[$id$] := votes[$id$] $\sqcup$ R[$id$];
    accepted := $\{id \in$ accepted : $|$votes[$id$]$| \geq n - t\}$;

    foreach $id \in$ accepted do
        for $|$votes[$id$]$|$ + 1 to $n$ do //fill missing votes with valid vote
            votes[$id$] := votes[$id$] $\sqcup$ my_ranks[$id$];
        for 1 to t do // remove $t$ extreme values
            votes[$id$] := votes[$id$] $\setminus$ {MAX(votes[$id$])};
            votes[$id$] := votes[$id$] $\setminus$ {MIN(votes[$id$])};
        SORT(votes[$id$]);
        new_ranks[$id$] := AVG(SELECT$_t$(votes[$id$]);
    return new_ranks;
```

to ensure that the approximation of the validated values will still be done in a consistent way.

In addition to the variables and functions introduced before, the second phase of our algorithm also needs the following data structures and auxiliary functions: variable R is a set of *ranks* arrays; the function ROUND(x) returns the integral value nearest to x; finally, the function SELECT$_k$(*set*) returns a choice of values from a set. These values are chosen to maximize the convergence rate of the approximate agreement. Later in the text we describe what is the most appropriate choice function.

In detail, each voting round works as follows. Processes exchange the values in their *ranks* array. Each array received from a remote process is first validated as described earlier. If the array is considered valid, it is added to the set of votes received in the current round. At the end of the round, votes are processes by the function APPROXIMATE, depicted in Algorithm 7. In this function, each process computes a new rank for each id in the *accepted* set as follows. It first collects all votes received for a given id into multiset votes[id] (multiset is a set that allows repetitions). If for some id in *accepted*, less than $n - t$ votes are received, this id is discarded (by construction, this never happens to an id that has been considered timely by some correct process). For the remaining ids, if the number of votes is less than n, the process fills the multiset by including copies of its own value (local values of a correct process are always valid). Then, the resulting multiset of n votes is sorted and the t lower values and t higher values are discarded. Finally, function $\textsc{select}_t$ is used to pick a subset from the remaining values that is averaged to compute the new vote for that id. This function returns a multiset consisting of each $(it+1)$th element of the *set* (which is an ordered multiset), where $0 \le i < \lfloor \frac{|set|}{t} \rfloor$. In other words, $\textsc{select}_t(set)$ returns a multiset consisting of the smallest and each t-th element after it. This choice of $\textsc{select}_t$ is the same as in the approximate agreement algorithm of [DLP$^+$86], which guarantees the convergence rate of $\sigma_t = \lfloor \frac{n-2t}{t} \rfloor + 1$ where σ_t is a number of elements returned by $\textsc{select}_t$.

After executing $2 \log t + 5$ approximation rounds, the new name is chosen as the rounded value of rank[my_id]. The stretch factor of δ applied to the inputs and the validation procedure ensures that the ranks converge preserving a distance of slightly more than 1, which prevents the rounded ranks from clashing in spite of a possible approximation error.

6.1.3 Correctness

We start by stating that any id in *timely* at some correct process, is necessarily included in *accepted* of any other correct process.

Lemma 27. *For any id such that $id \in timely_p$ at some correct p, then $id \in accepted_q$ at any correct q.*

Proof Assume by contradiction, $id \notin accepted_q$ at some correct q. This is only possible if q has not received $n - 2t$ $\langle \textsc{Ready}, \text{id} \rangle$ messages in Rounds 3 (Lines 17-18 of Algorithm 5).

But if p added *id* into *timely*, it means that it has received at least $n - t$ $\langle \textsc{Ready}, \mathrm{id} \rangle$ messages, $n - 2t$ of which must have been sent by the correct processes in Round 3. Therefore all correct processes have received at least $n - 2t$ $\langle \textsc{Ready}, \mathrm{id} \rangle$ messages in Round 3, which leads to a contradiction. □

The following lemma states that all correct ids are included in *timely* sets of all correct processes.

Lemma 28. *If id belongs to some correct p, then id* $\in$ *timely*$_q$ *at any correct q.*

Proof Assume by contradiction, $id \notin timely_q$ for some correct q. This means that q has not received $n - t$ $\langle \textsc{Ready}, \mathrm{id} \rangle$ in Round 3. This is only possible if some correct process has not issued $\langle \textsc{Ready}, \mathrm{id} \rangle$, which in turn is because it has not received $n - t$ $\langle \textsc{Echo}, \mathrm{id} \rangle$ in Round 2. This also is only possible if *id* was not received by some correct process in Round 1. However, since p is correct, p sent *id* to all correct processes in Round 1. Contradiction. □

The following lemma will be used to calculate the maximum number of ids that Byzantine processes are able to produce.

Lemma 29. *If id* $\in$ *accepted*$_p$ *at some correct p, then at least* $n - 2t$ *correct processes received id in Round* 1.

Proof If $id \in accepted$, then p has received at least $n - 2t$ $\langle \textsc{Ready}, id \rangle$ messages from which at least 1 must have been issued by a correct process. This means that some correct process received at least $n-t$ $\langle \textsc{Echo}, id \rangle$ messages in Round 2, $n-2t$ of which must have come from the correct processes. □

As discussed earlier, Byzantine processes can generate more than t identifiers, none of which recognized as faulty by the correct processes. The following lemma bounds the total number of ids included in *accepted* at any correct p.

Lemma 30. *At the end of Round* 3*, if p is correct, then*

$$|accepted_p| \leq n + \left\lfloor \frac{t^2}{n - 2t} \right\rfloor .$$

Proof By Lemma 28, all $n-t$ correct ids are in $timely_p$, therefore also in $accepted_p$. It remains to calculate the maximum number of Byzantine ids that can be in $accepted_p$. By Lemma 29, each $id \in accepted_p$ must have been echoed in Round 2 by at least $n-2t$ correct processes. This means that from the total of at most $t(n-t)$ identifiers broadcast by the Byzantine processes in Round 1, $\lfloor \frac{t(n-t)}{n-2t} \rfloor = t + \lfloor \frac{t^2}{n-2t} \rfloor$ can be in $accepted_p$ at the end of Round 4. □

The following lemma is auxiliary and states that if we construct two multisets by adding pairwise values separated by some given distance from each other, then after we order the multisets, the entries on the corresponding indexes still preserve this distance.

Lemma 31. *Let U and W be two ordered multisets with k elements each, created by adding k pairs of elements $a, pair(a)$ into U, W respectively, such that $a + \delta \leq pair(a)$. Then, for any $1 \leq i \leq k$, $u_i + \delta \leq w_i$.*

Proof We first show that the inequality holds for the first elements in the ordered multisets, i.e.

$$u_1 + \delta \leq w_1. \tag{6.1}$$

Since w_1 is the smallest in W, $w_1 \leq pair(u_1)$. If $w_1 = pair(u_1)$, then (6.1) follows. If $w_1 < pair(u_1)$, there exists u_i such that $w_1 = pair(u_i)$. Since u_1 is the smallest in U, $u_1 + \delta \leq u_i + \delta \leq w_1$, as claimed.

Now, by making $pair(u_1)$ a new pair of u_i, the same argument is used to iteratively prove (6.1) for $U = U \setminus \{u_1\}$ and $W = W \setminus \{w_1\}$ until U and W are empty. Therefore, $1 \leq i \leq k$, $u_i + \delta \leq w_i$, as needed. □

The following lemma shows that during the approximation procedure, the distance between the ranks of two ids included in the *timely* set of some correct process maintains at least δ.

Lemma 32. *If for some ids $id, id' \in timely$, at the beginning of Round r, $ranks[id] + \delta \leq$*

$ranks[id']$ and $|votes[id]|, |votes[id']| \geq n - t$, then at the end of Round r, $ranks[id] + \delta \leq ranks[id']$.

Proof Since $id, id' \in timely$, all votes accepted in Line 25 must contain new ranks for both id and id' spaced by at least δ. Hence, $|votes[id]| = |votes[id']|$.

If there are less than n entries in each set, the $ranks[id]$ and $ranks[id']$ will be added respectively such that both sets have exactly n entries (Lines 10-11 of Algorithm 7), (by assumption, the added values also preserve the distance of at least δ).

Now, assume U, W are multisets resulted from ordering $votes[id]$ and $votes[id']$ respectively. By Lemma 31, for any $1 \leq i \leq n$, $u_i + \delta \leq w_i$. Hence, after deleting from U and W, t smallest and t largest entries (Line 13-14 of Algorithm 7), it still holds that $1 \leq i \leq n - 2t$, $u_i + \delta \leq w_i$. The distance between the new values (calculated in Line 16) is given by,

$$\begin{aligned} & \textsc{avg}(\textsc{select}_t(W)) - \textsc{avg}(\textsc{select}_t(U)) \\ \geq \quad & \frac{\textsc{sum}(\textsc{select}_t(U)) + t\delta}{t} - \frac{\textsc{sum}(\textsc{select}_t(U))}{t} \\ = \quad & \delta. \end{aligned}$$

□

We then show that correct processes always issue valid votes.

Lemma 33. *For any $r \geq 4$, if $ranks_p$ and $ranks_q$ are held by any two correct p and q in Round r, then*

$$\textsc{isValid}(ranks_p, ranks_q) = true.$$

Proof $\textsc{isValid}(ranks_p, ranks_q)$ checks if the distance between the ranks of all elements in $timely_p$ is at least δ. By Lemma 27, $timely_q \subseteq accepted_p$. Therefore, if the entries in $ranks_p$ preserve the distance of least δ, for any id such that $id \in \bigcup_{q:\ q\ is\ correct} timely_q$, in Round r, then $\textsc{isValid}(ranks_p, ranks_q)$.

We now show by induction on r that the distance between the ranks of ids in $timely_p$ is preserved at least δ by all correct processes in any Round $r \geq 5$. For the base case of $r = 5$, recall that p constructs the initial ranks in such a way that all ranks for the *accepted* set are spaced by at least δ (Line 28 of Algorithm 5), therefore $\textsc{isValid}(ranks_p, ranks_q) = true$.

For the induction round, assume that, for the *rank* held by p in Round r, $\textsc{isValid}(ranks_p, ranks_q) = true$. Therefore, for each element in *timely* each correct process will receive at least $n - t$ valid votes. And since by assumption, the correct votes are valid in Round r and by Lemma 28 each correct vote contains new ranks for all ids in $timely_p$, p will update their values in Line 35 of Algorithm 5 and, by Lemma 32, the new ranks calculated by each correct process at the end of Round r preserve the necessary distance at least δ. Therefore, $\textsc{isValid}(rank_p, rank_q) = true$ in $r + 1$. □

Corollary 34. *If $id \in timely_p$ at some correct p, then its rank is updated in every approximation round by each correct process.*

Corollary 35. *If $id < id'$ belong to two correct processes, then*

$$ranks_p[id] + \delta \leq ranks_p[id'],$$

at any correct p in every Round $r \geq 3$.

We now need to bound the maximum discrepancy in the initial ranks for the same ids.

Lemma 36. *If $id \in timely_p$ for some correct p, then at the end of Round 3,*

$$|ranks_p[id] - ranks_q[id]| \leq (t + \lfloor \frac{t^2}{n - 2t} \rfloor) \times \delta,$$

where $ranks_q[id]$ is the rank of id at some correct q.

Proof By assumption, $id \in timely_p$, therefore, by Lemma 27, $id \in accepted_q$. Also, by Lemma 28, all correct ids are in $timely_p$ and $timely_q$ and therefore in *accepted* at each correct process. Hence, $|accepted_p \cap accepted_q| \geq n - t$. On the other hand, by Lemma 30, all correct processes have $|accepted| \leq n + t - 1$. Hence, the initial ranks calculated in

Line 28 of Algorithm 5 of each common element of $accepted_p$ and $accepted_q$ differs by at most $(2t-1) \times \delta$. $\square$

Now it remains to show that each approximation round of Algorithm 7 reduces the distance between the ranks by the factor $\sigma_t = \lfloor \frac{n-2t}{t} \rfloor + 1$.

Lemma 37. *Let id $\in$ timely$_p$ at some correct p, and Δ_r denote the maximum distance between the correct ranks for id in the beginning of Round r. Then, at the end of Round r, the distance between new correct ranks for this id is within the range of $\frac{\Delta_r}{\sigma_t}$. Moreover, the new values are within the range of the old values belonging to correct processes.*

Proof Since $id \in timely_p$, then by Lemma 33 and Corollary 34, $votes_p[id]$ and $votes_q[id]$ have at least $n-t$ entries from the correct processes, therefore after executing Lines 12-14 of Algorithm 7 both multisets have exactly n entries.

Let C be the multiset of ranks of id issued by all correct processes in Algorithm 5, in Round r. Note that $C \subseteq votes_p[id], votes_q[id]$.

Let A, B be ordered multisets resulting from deleting t maximal values and t minimal values from $votes_p[id]$ and $votes_q[id]$, respectively. Let $a_1 \leq \cdots \leq a_c$ be the elements of $\textsc{select}_t(A)$ and $b_1 \leq \cdots \leq b_c$ be the elements of $\textsc{select}_t(B)$, where c is the number of elements selected. Note that $c = \sigma_t$.

First, we need to show that, for $1 \leq i \leq c-1$,

$$\textsc{max}(a_i, b_i) \leq \textsc{min}(a_{i+1}, b_{i+1}). \tag{6.2}$$

It suffices to show that $a_i \leq b_{i+1}$, then by symmetric argument $b_i \leq a_{i+1}$. Suppose, by contradiction, that $a_i > b_{i+1}$. There are at least $t(i+1)+1$ elements in B less than or equal to b_{i+1}. By our supposition, these elements are strictly less than a_i. However, there are at most ti elements in A strictly less than a_i. Therefore, at least $t(i+1)+1-ti = t+1$ elements in B, are not in A. However, since $|votes_p[id] \cap votes_q[id]| \geq n-t$, it holds that

$|A \cap B| \geq n - t - 2t$. Therefore,

$$|B - A| = |B - (A \cap B)| \leq (n - 2t) - (n - 3t) = t.$$

Hence the contradiction and (6.2) follows.

We then use (6.2) to prove the lemma. The discrepancy between $ranks_p[id]$ and $ranks_q[id]$, which are updated in Line 16 of Algorithm 7 at the end of Round r, is given by,

$$\begin{aligned}
& |\textsc{avg}(\textsc{select}_t(A)) - \textsc{avg}(\textsc{select}_t(B))| \\
= \quad & \frac{1}{c}\left|(a_1 + \cdots + a_c) - (b_1 + \cdots + b_c)\right| \\
= \quad & \frac{1}{c}\left|\sum_{i=1}^{c}(a_i - b_i)\right| \\
\leq \quad & \frac{1}{c}\sum_{i=1}^{c}|a_i - b_i| \\
= \quad & \frac{1}{c}\sum_{i=1}^{c}\left(\textsc{max}(a_i, b_i) - \textsc{min}(a_i, b_i)\right),
\end{aligned} \tag{6.3}$$

where the fourth line follows from triangular inequality.

Expanding the sum and successively applying (6.2),

$$\begin{aligned}
& \frac{1}{c}\sum_{i=1}^{c}\left(\textsc{max}(a_i, b_i) - \textsc{min}(a_i, b_i)\right) \\
= \quad & \frac{1}{c}\left(\textsc{max}(a_c, b_c) - \textsc{min}(a_c, b_c)\right) \\
+ \quad & \frac{1}{c}\sum_{i=1}^{c-1}\left(\textsc{max}(a_i, b_i) - \textsc{min}(a_i, b_i)\right) \\
\leq \quad & \frac{1}{c}\left(\textsc{max}(a_c, b_c) - \textsc{min}(a_1, b_1)\right).
\end{aligned} \tag{6.4}$$

On the other hand, since we deleted t extremal values from $votes_p[id]$ and $votes_q[id]$, it is true that $\textsc{max}(a_c, b_c) \leq \textsc{max}(C)$ and $\textsc{min}(a_1, b_1) \geq \textsc{min}(C)$. Therefore, the averages are within the interval of the input values belonging to the correct processes.

Moreover, from (6.3) and (6.4),

$$\begin{aligned}
&|\text{AVG}(\text{SELECT}_t(A)) - \text{AVG}(\text{SELECT}_t(B))| \\
\leq\ & \frac{1}{c}\left(\text{MAX}(C) - \text{MIN}(C)\right) \\
=\ & \frac{1}{\sigma_t}\Delta_r.
\end{aligned}$$

Hence, the lemma follows. □

We now calculate the number of iterations needed to reduce Δ_r to less than $\frac{1-\delta}{2}$.

Lemma 38. *If $\Delta_4 \leq (2t-1) \times \delta$, then after $r = 2\lceil \log t \rceil + 5$ iterations, the range of the values belonging to all correct processes is less than $\Delta_{r+4} < \frac{\delta-1}{2}$.*

Proof By successive applications of Lemma 37,

$$\begin{aligned}
\Delta_{r+4} &\leq \left(\frac{1}{\sigma_t}\right)^r \Delta_5 \\
&< \left(\frac{1}{2}\right)^{\lceil 2\log(t)\rceil + 5} 2t \times \left(1 + \frac{1}{n+t}\right) \\
&< \frac{1}{6\,(n+t)}.
\end{aligned}$$

□

Finally, we are ready to prove the main theorem.

Theorem 39. *Algorithm 5 solves order-preserving renaming with $n > 3t$ and target namespace of size $n + t - 1$.*

Proof

Validity. By Lemma 30, if p is correct, $|accepted_p| \leq n + \lfloor \frac{t^2}{n-2t} \rfloor \leq n + t - 1$, for $n > 3t$. Therefore, the initial ranks are bounded by $(n+t-1) \times \delta$. Since by Lemma 37, all correct processes output a value within the interval of the initial correct values, the outputs of the correct processes are bounded by $\text{ROUND}((n + t - 1) \times \delta) = n + t - 1$.

Termination. After $2\lceil \log t \rceil + 8$ rounds, every correct process outputs a value.

Order-preserving property. By Lemma 28, correct ids are always included in *timely* sets and, by Corollary 34, are updated in each round by every correct process. By Corollary 35, for any two correct id and id' such that $id < id'$, the distance between their ranks is lower bounded by δ in every round. Since by Lemma 38, after $2\lceil \log t \rceil + 8$ rounds, $\Delta_r < \frac{\delta - 1}{2}$,

$$rank\,(id) + \delta + \frac{1-\delta}{2} < rank\,(id') - \frac{1-\delta}{2}.$$

Hence, $\textsc{Round}(ranks[id])) < \textsc{Round}(ranks[id'])$. □

6.1.4 Complexity Analysis

By construction, the round complexity of Algorithm 5 is $2\lceil \log t \rceil + 8$. In each round, processes employ all-to-all communication. As a result, the total message complexity is $O\left(n^2 \log t\right)$. Since in each round processes exchange arrays of at most $n+t-1$ original ids and their ranks, the message size is bounded by $O\left((n+t-1)\left(\log N_{max} + \log n\right)\right)$ bits.

6.2 Algorithm for $n > t^2 + 2t$ with 7 Rounds and Tight Namespace

An interesting property of Algorithm 5 is that it performs tight renaming, i.e. renaming with the target namespace of size n, and can terminate after a constant number of rounds if $n > t^2 + 2t$. The optimal namespace is achieved because Byzantine processes are not able to introduce any additional identifiers in our id selection scheme. The constant round complexity is due to the fast convergence of Byzantine AA. Similar argument was used by the authors of [AAGT12] to prove the constant round complexity of the crash-tolerant algorithm presented in [Oku10] when the number of actual crashes is bounded by $n > 2f^2$.

This result is formalized below.

Lemma 40. *If $n > t^2 + 2t$, Algorithm 5 achieves the target namespace of size n.*

Proof By Lemma 30, the number of ids in the *accepted* set of any correct process is at most $n + \lfloor \frac{t^2}{n-2t} \rfloor = n$. Due to the stretching factor of $\delta = 1 + \frac{1}{3(n+t)}$, the initial ranks are bounded by $n \times \delta$. Since by Lemma 37 the values returned by the approximation belong to the interval of the initial correct values, the rounded outputs will be at most $\textsc{round}(n \times \delta) = n$. $\square$

Lemma 41. *If $n > t^2 + 2t$, after 4 approximation rounds, the values held by the correct processes are within the distance of less than $\frac{\delta-1}{2} = \frac{1}{6(n+t)}$.*

Proof By Lemma 36, the maximum discrepancy between the votes is at most $(t + \lfloor \frac{t^2}{n-2t} \rfloor) \times \delta = t \times \delta$. On the other hand, by Lemma 37, the convergence rate of each approximation round is at least $\sigma_t = \lfloor \frac{n-2t}{t} \rfloor + 1 > \lfloor \frac{t^2}{t} \rfloor + 1 = t + 1$. Therefore, after 4 convergence rounds, the values of the correct processes are within

$$\frac{t \times \delta}{(t+1)^4} < \frac{1}{3t^3} < \frac{\delta - 1}{2}.$$

$\square$

Therefore, if we change the code of Algorithm 5 to run only 4 approximation rounds (Line 29), the resulting algorithm has the complexity of 7 rounds.

Theorem 42. *The modified algorithm solves tight order-preserving renaming in $O(1)$ rounds if $n > t^2 + 2t$.*

6.3 Algorithm for $n > t^2 + 2t$ with Only 2 Rounds

In the previous section we have shown that the modified Algorithm 5 has constant round complexity with $n > t^2 + 2t$. This is an interesting result from the asymptotic point of view, specially considering that the resulting name space is optimal. Still, from the practical point of view, the number of communication rounds can still be an impairment for time constrained applications (the number of rounds of the modified Algorithm 5 is exactly 7). Therefore, in this section we are interested in performing renaming in as few communication rounds as possible. Interestingly, we show that order-preserving renaming in face of Byzantine faults can be solved in just 2 communication rounds with $n > 2t^2 + t$, by relaxing the target namespace to n^2. Obviously, in just 2 communication rounds, it is impossible to perform iterative approximate agreement. In fact, our algorithm is simply based on counting echoes that are filtered by a validity check.

The algorithm is depicted in Algorithm 8. The main idea of the algorithm is having each process initially announce its ids to all other processes; then, echo all the ids received in the first round, and finally having each correct process calculate its new name by ordering all the received ids, and calculating *offsets*, i.e. spacings between two consecutive names, according to the number of echoes received for each id. Byzantine processes may opt not to echo the ids or even send contradictory information to different processes. Therefore, correct processes may receive different sets of ids as well as different numbers of echoes for each ids. The key to the algorithm is to compute the offsets in such a way that the new names chosen by the correct processes will hold the order-preserving property, despite potentially inconsistent sets of echoes.

As the previous algorithms, Algorithm 8 also uses a *timely* and an *accepted* set of ids. In this algorithm, all ids broadcast in Round 1 are considered timely and all ids echoed in Round 2, that pass a basic validity test, are accepted. The validity test, captured by function isValid, limits the power of Byzantine processes as follows: first, it only accepts echo messages from processes that have sent their id in Round 1; then, it does not accept a MultiEcho message that has more than n ids; finally, the incoming MultiEcho must have at least $n - t$ ids in common with the timely set of the recipient (note that if the sender and the recipient of a MultiEcho are correct, they both have at least $n - t$ correct ids in their timely sets). Also, for each accepted id, the algorithm counts how many processes have echoed that id (again, correct ids are guaranteed to be echoed at least $n - t$ times).

After all echo messages have been processed, processes are ready to calculate new names. The offset for each known id is simply the value of min($counter, n - t$) (Line 20).

Algorithm 8 2-round Order-preserving Byzantine Renaming for $n > 2t^2 + t$

```
01 Init:
02     foreach lnk ∈ $\{1, \cdots, n\}$ linkid[$lnk$] :=$\perp$;
03     timely := accepted := $\emptyset$;
04     forall $id$ do counter[$id$] := 0; // init sparse array with zeros

05 In Round $r$ := 1
06     broadcast ($\langle$ID, $my\_id\rangle$);
07     foreach $id$: $\langle$ID, $id\rangle$ received from a distinct link $lnk$ do
08         linkid[$lnk$] := $id$;
09         timely := timely $\cup$ $\{id\}$;

10 In Round $r$ := 2
11     broadcast ($\langle$MultiEcho, timely$\rangle$);
       // count echoes
12     foreach $id$: $\langle$MultiEcho, ids$\rangle$ received from a distinct link $lnk$ do
13         if isValid ($lnk$, $ids$) then
14             foreach $id \in$ ids do
15                 accepted:= accepted $\cup$ $\{id\}$;
16                 counter[$id$] := counter[$id$] +1;
       // compute new names
17     sort (accepted);
18     accum_offset := 0;
19     for $id$ := first(accepted) to last(accepted) do
20         accum_offset := accum_offset + min (counter[$id$], $n - t$);
21         newid[$id$] : = accum_offset;
22     return newid[$my\_id$]


01 Function isValid (lnk, ids) returns boolean is
02     return (linkid[lnk] $\neq\perp$) $\wedge$ ($|$ids$| \leq n$) $\wedge$ ($|$timely $\cap$ ids$| \geq n - t$)
```

The adjustment to $n - t$ guarantees that these offsets for the correct ids are always the same. This prevents Byzantine processes from introducing an additional error linear in the number of correct processes by choosing to echo correct ids for some processes but not others. Finally, the new name of a process is produced by summing the offsets of all ids up to, and including, the id of the current process. The algorithm also stores locally estimated values of new names for other processes (Line 20); this is not required in practice and is done here only for clarity of the proofs.

6.3.1 Correctness

Let Δ denote the maximum possible discrepancy between the new names for some correct id.

Lemma 43. $\Delta \leq 2t^2$.

Proof For each echo message received in Round 2, a correct process compares the number of ids in common, that should be at least $n - t$ out of n allowed per message (procedure isValid). Due to this sanity check, each Byzantine process can introduce only $2t$ Byzantine ids in an echo message: in the worst case, the Byzantine process includes t Byzantine ids already known to the receiver and additional t Byzantine ids. Therefore, the total number of echoes of Byzantine ids received from the Byzantine processes by each correct process in Round 2, is at most $2t^2$. $\square$

We now need to lower bound the offset of any correct id.

Lemma 44. *Let id and id′ be two correct identifiers. If $id' < id$, then $newid_p[id'] + (n - t) \leq newid_p[id]$ at some correct p.*

Proof Assume, by contradiction, $newid_p[id] - newid_p[id'] < n - t$. This is only possible if $counter_p[id] < n - t$ (Line 20). This means that, in Round 2, p received less than $n - t$ echoes of *id*. It can only happen if some correct p' did not echo *id*. This, in turn, is only possible if p' did not receive *id* in Round 1. But since *id* is correct, it was sent to all the processes in Round 2. Contradiction. $\square$

We are ready to prove the correctness.

Theorem 45. *Algorithm 8 solves order-preserving renaming with $n > 2t^2 + t$ and the target namespace of size n^2.*

Proof

Validity. The total number of echoed ids accepted by each correct process in Round 2 is at most n^2. Therefore, the correct processes output an integer value within the range $[1, \cdots, n^2]$. Hence, Algorithm 8 satisfies the validity property.

Termination. After 2 rounds, every correct process outputs a value.

Order-preserving property. Consider two correct processes p and q with initial identifiers id and id', such that $id < id'$. By Lemma 44, $newid_p[id] + n - t \leq newid_p[id']$. Since by Lemma 43, $\Delta \leq 2t^2$, then $newid_p[id'] - 2t^2 \leq newid_q[id']$. Furthermore, since $n > 2t^2 + t$, $newid_p[id] + n - t - 2t^2 < newid_q[id']$. □

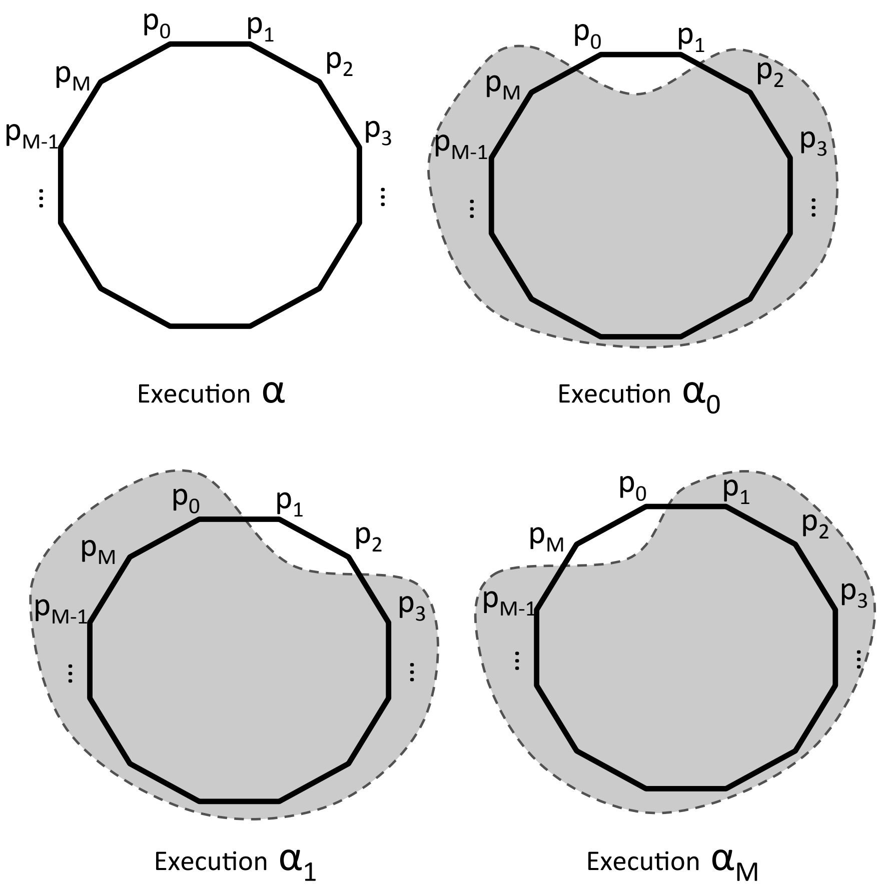
6.3.2 Complexity Analysis

Algorithm 8 consists of 2 communication rounds. Since in each round, processes employ all-to-all communication, the total message complexity is $2n^2$. In Round 2, processes exchange vectors of all ids they have received in Round 1. Therefore, the message size is bounded by $O(n \log N_{max})$ bits.

6.4 Impossibility Result for $n \leq 3t$

In previous sections we have seen that order-preserving renaming can be solved efficiently for different values of t smaller than $n/3$. We now address the question whether it is possible to solve this problem for $t \geq n/3$, as is the case with the original renaming problem (even though it requires a very inefficient algorithm). We establish the separation result for order-preserving renaming by showing that there is no deterministic algorithm with $n \leq 3t$ that solves order-preserving renaming. Our proof method is based on the indistinguishability argument widely used in the literature on Byzantine fault tolerance, e.g. [FLM85, KY86, GY89].

We first give the impossibility proof for deterministic algorithms in the system with $n = 3$ and $t = 1$. In the proof, we consider the easiest case of order-preserving renaming from a *bounded* original namespace of size $M + 1$ into a target namespace of size M, for some



Area in gray corresponds to a system simulated by a single Byzantine process.

Figure 6.1: Indistinguishable executions for $n = 3$ and $t = 1$.

arbitrary $M \geq n$. This implies *a fortiori* that the impossibility applies to namespaces of any size, as long as the original namespace is larger than the target namespace (otherwise the problem becomes trivial).

We take a candidate algorithm for the case $n = 3$ and construct an indistinguishability ring of executions, which violates the properties of order-preserving renaming. Namely, we construct a ring of size $M + 1$ and assign inputs in such a way that new names must increase as we traverse the ring in one direction, but this exhausts the target namespace after going around the ring.

Theorem 46. *There is no deterministic algorithm that solves order-preserving renaming in a system with $n = 3$ and $t = 1$.*

Proof Assume by contradiction there exists an algorithm π that solves order-preserving renaming for $n = 3$ and $t = 1$ from the original namespace of size $M + 1$ into the target namespace of size M, for some $M \geq 3$. Without loss of generality we assume π is a full-information algorithm.

Consider a system with $M + 1$ processes $p_0, \ldots, p_M$. We consider an execution α of π in this system, where processes are arranged in a ring as depicted in Fig. 6.1, and process p_i has original id i. This is an execution with $M + 1 > n$ processes, even though algorithm π is designed for $n = 3$ processes, but we will argue this execution has some interesting properties. For every pair of adjacent processes in the ring, their view of the system is indistinguishable from the view in a 3-process system in which the two processes are connected to a corrupted third process. For instance, processes p_0 and p_1 cannot distinguish execution α from execution α_0 in Fig. 6.1 where both p_0 and p_1 are connected to a single corrupted process that simulates p_M and p_2. That is, the single Byzantine process sends to p_0 exactly what p_M sends to p_0 in α, and the same Byzantine process sends to p_1 exactly what p_2 sends to p_1 in α (in Fig. 6.1, the gray area in execution α_0 depicts the system simulated by a single Byzantine process). By assumption, in α_0 processes p_0 and p_1 decide on valid names in the correct order. Moreover, since p_0 and p_1 cannot distinguish execution α from execution α_0, they decide in α exactly on the same names as in α_0. Similarly, p_1 and p_2 cannot distinguish α from α_1 in Fig. 6.1, and so on for each pair of adjacent processes in the ring. Therefore, in α, each pair of adjacent processes decides on valid names in the correct order.

By the validity property, in α process p_0 decides on a new name v_0 such that $1 \leq v_0 \leq M$. By the order-preserving property, p_1 decides on name v_1 such that $1 \leq v_0 < v_1$. Applying the order-preserving property to the new names of all pairs of processes from p_0 to p_M, we see that process p_M decides on a new name v_M such that $1 \leq v_0 < v_1 < \ldots < v_{M-1} < v_M$. Thus, $v_M > M$. But by the validity property, $v_M \leq M$—a contradiction to the existence of algorithm π. $\square$

Theorem 46 is generalized to the case of $t = \lceil n/3 \rceil$ by having three processes simulate the n-process system as described in [OBG08]. More precisely, if there is an algorithm π

for n processes and $t = \lceil n/3 \rceil$, we can use π to obtain an algorithm for three processes and $t = 1$, which contradicts Theorem 46. The simulation algorithm for three processes works as follows. Each process simulates $\lceil n/3 \rceil$ or $\lfloor n/3 \rfloor$ processes running algorithm π, for a total of n simulated processes, where the simulated processes start with names that respect the id ordering of the three processes; each process then decides on the new name of any of the processes that it simulates.

Theorem 46 concerns deterministic algorithms. We now consider *randomized* algorithms. We use the indistinguishability argument to show that any randomized algorithm with $n = 3$ has a non-zero probability of error when running in the system depicted in Fig. 6.1. Again, we assume the easiest case of order-preserving renaming from the original namespace of size $M + 1$ into the target namespace of size M, for some arbitrary $M \geq n$. In our proof, we assume the non-rushing adversary, which is weaker than the rushing counterpart. Therefore, the impossibility holds under both adversaries.

Theorem 47. *Under the non-rushing adversary, there is no randomized algorithm that solves order-preserving renaming in a system with $n = 3$ and $t = 1$.*

Proof Assume that there exists a (full information) randomized algorithm π' that solves order-preserving renaming for $n = 3$ and $t = 1$ from the original namespace of size $M + 1$ into the target namespace of size M, for some $M \geq 3$.

Consider a system composed by $M + 1$ processes $p_0, \ldots, p_M$ arranged in a ring as depicted in Fig. 6.1. We show that there exists a finite execution in the ring such that all $M + 1$ processes terminate. From that point, the proof proceeds as in Theorem 46. The proof of the existence of such execution is slightly technical but follows from the termination with probability 1 of π'. The proof will construct increasingly larger executions in which, successively, processes $p_0, \ldots, p_M$ terminate, by arguing for each process p_i that it would terminate with probability 1 in a 3-process system.

More precisely, for each pair of processes p_i and p_{i+1} the execution in the ring is indistinguishable from an execution α_i where p_i and p_{i+1} is connected to a single Byzantine process that behaves exactly like p_{i-1} and p_{i+2} in α (the arithmetic of indices is done modulo $(M+1)$). From the termination with probability 1 property of order-preserving renaming, (*) all correct processes in a 3-process system running π' with $t=1$ terminate with probability 1. We now consider executions of π' in the ring of $M+1$ processes. We first claim that there is a finite execution β_0 of π' in the ring such that p_0 and p_1 terminate. This follows from (*) and the fact that an execution in the ring is indistinguishable by p_0 and p_1 from an execution in a 3-process system. We now claim that we can extend execution β_0 such that p_2 also terminates. Indeed, β_0 is indistinguishable by p_1 and p_2 from an execution in a 3-process system. Since β_0 is finite, it occurs with positive probability $p_0 > 0$. If p_2 never terminates in any extensions of β_0, we can find a set of executions with probability $p_0 > 0$ where p_2 never terminates, contradicting (*). Therefore, in some extension of β_0, p_2 terminates. Let β_1 be the finite execution that combines β_0 and this extension until p_0, p_1, and p_2 terminate. We apply the same argument with execution β_1 and process p_3 to obtain a finite execution β_2 where $p_0, \ldots, p_3$ terminate. We continue this construction with all other processes, to obtain a finite execution β_{M-1} where all processes $p_0, \ldots, p_M$ terminate. $\square$

Theorem 47 is generalized to the case of $t = \lceil n/3 \rceil$ by the simulation by 3 processes of an n-process system given in [OBG08], as previously described for the deterministic algorithm.

Chapter 7

Conclusions

This thesis presents a number of results on different aspects of renaming and order-preserving renaming problems. In particular, we have focused on renaming in the classical synchronous message-passing model and considered both crashes and byzantine faults. The results, summarized in Table 1.1 and Table 1.2, support our hypothesis that renaming can be solved both efficiently and with high resiliency in these settings. Many of our results have been achieved with resort to randomization. This implies that randomization is a powerful and necessary technique in solving renaming.

When considering crash faults, we have shown that tight renaming can be solved very efficiently by using randomization. We prove this by proposing a randomized algorithm that we call Balls-into-Leaves due to its connection to the classical balls-into-bins technique. The algorithm places n balls into n leaves of a binary tree in $O(\log \log n)$ rounds whp. Our simulations suggest that the hidden constants in our asymptotic analysis are indeed small. In fact, the algorithm terminates in few rounds even for very large values of n. We do not optimize the algorithm for the message complexity—in each round Balls-into-Leaves employs all-to-all communication. Therefore, the total message complexity is $O(n^2 \log \log n)$ whp. It is possible to reduce this value to $O(n^2)$ whp by eliminating communication between balls in disjoint subtrees. However, overcoming the $O(n^2)$ barrier ($O(n)$ messages per ball) would require a substantially different approach.

We have also presented an extension of the algorithm that provides early termination in $O(\log\log f)$ rounds whp when there are f failures. The modified algorithm terminates optimally in $O(1)$ rounds in failure-free executions.

These results imply an exponential separation between deterministic and randomized algorithms for tight renaming. The main open question is whether the Balls-into-Leaves algorithm is optimal for this problem. Answering this question requires new lower bounds for randomized renaming. We conjecture that obtaining such lower bounds will be challenging, given that lower bounds for other variants of renaming have required subtle topological or reduction techniques, e.g. [HS99, CnR08, Gaf09, AACH+14].

We then turn our attention to a more challenging failure model and consider Byzantine faults. In particular, we have shown that tight renaming can be solved efficiently in the presence of Byzantine faults under the assumption of the *non-rushing* adversary. Our algorithm terminates in $O(\log n)$ rounds whp, and works for any $t < n$. An interesting open question is whether it is possible to solve this problem with sub-logarithmic round complexity. We have also shown that our algorithm works for $t = 1$ under the *rushing* adversary. For the general case of $t > 1$, we strengthened the algorithm with a cryptographic commitment scheme, which requires the additional assumption of a polynomially bounded adversary. Avoiding the use of cryptography for $t > 1$ is an interesting open problem.

Additionally, this thesis is the first to address order-preserving renaming in the Byzantine fault model. We show that this problem can also be solved efficiently in this context.

Our first order-preserving algorithm terminates in $O(\log n)$ rounds and tolerates up to $t < n/3$ Byzantine faults. Whether this round complexity is optimal for this problem is an open question. The algorithm has target namespace of size $n + t - 1$, presenting an improvement on the namespace size compared to the previous result for the Byzantine setting [OBG08] that does not ensure the ordering (the algorithm in [OBG08] ensures the target namespace of size at most $2n$). Interestingly, if $n > t^2 + 2t$, our algorithm achieves *tight* namespace. It remains open whether it is possible to achieve tight namespace for larger values of t.

If the resiliency is bounded by $O(\sqrt{n})$, we have shown that order-preserving renaming can be solved in constant time both by using approximate agreement and with a simple double-echo scheme. This resiliency threshold for constant round complexity asymptotically matches the results for the crash fault model [AAGT12]. We leave for future work to find out whether the $O(\sqrt{n})$ threshold is optimal or higher resiliency can be achieved in constant time.

Finally, we have proven a $t < n/3$ bound on the resiliency of both deterministic and randomized algorithms for order-preserving renaming. This result reveals a separation between the resiliency of renaming and order-preserving renaming. In other words, order-preserving renaming is provably a stronger problem than renaming. It will be interesting to find out whether randomization can help in reducing the round complexity of order-preserving renaming.

Bibliography

[AACH$^+$14] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *J. ACM*, 61(3):18:1–18:51, June 2014.

[AAGT12] Dan Alistarh, Hagit Attiya, Rachid Guerraoui, and Corentin Travers. Early deciding synchronous renaming in O(log f) rounds or less. In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity*, SIROCCO '12, Reykjavik, Iceland, June 2012.

[ABND$^+$90] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37:524–548, July 1990.

[ACMR95] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 238–247, New York, NY, USA, 1995.

[AF02] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, February 2002.

[BEW11] Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119–134, 2011.

[BG93] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Prin-*

ciples of Distributed Computing, PODC '93, pages 41–51, New York, NY, USA, 1993. ACM.

[BKSS13] Petra Berenbrink, Kamyar Khodamoradi, Thomas Sauerwald, and Alexandre Stauffer. Balls-into-bins with nearly optimal load distribution. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 326–335, Montreal, Canada, 2013.

[BN01] Rida A. Bazzi and Gil Neiger. Simplifying fault-tolerance: providing the abstraction of crash failures. *J. ACM*, 48:499–554, May 2001.

[BO83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[Bra87] Gabriel Bracha. An o(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, October 1987.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.

[CHT99] Soma Chaudhuri, Maurice Herlihy, and Mark Tuttle. Wait-free implementations in message-passing systems. *Theoretical Computer Science*, 220(1):211–245, June 1999.

[CnR08] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology upper and lower bounds for renaming. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 295–304, Toronto, Canada, 2008. ACM.

[CnR12] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *J. ACM*, 59(1):3:1–3:49, March 2012.

[CR10] Armando Castaeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.

[Dij82] Edsger W. Dijkstra. On weak and strong termination. In *Selected Writings on Computing: A Personal Perspective*, pages 355–357. Springer-Verlag, 1982.

[DLP$^+$86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33:499–516, May 1986.

[DS82] Danny Dolev and Raymond Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, STOC '82, pages 401–407, San Francisco (CA), USA, 1982.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[FL87] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, January 1987.

[FLM85] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, PODC '85, pages 59–70, New York, NY, USA, 1985. ACM.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[FM88] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 148–161, New York, NY, USA, 1988. ACM.

[Gaf09] Eli Gafni. The extended bg-simulation and the characterization of t-resiliency. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC '09, pages 85–92, Bethesda, Maryland, USA, 2009. ACM.

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.

[Gon81] Gaston Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, April 1981.

[GT89] Ajei Gopal and Sam Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 110–123. Springer Berlin Heidelberg, 1989.

[GY89] Ronald L. Graham and Andrew Chi-Chih Yao. On the improbability of reaching byzantine agreements (preliminary version). In *STOC*, pages 467–478, 1989.

[HS99] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999.

[IR81] Alon Itai and Michael Rodeh. Symmetry breaking in distributive networks. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:150–158, 1981.

[KPP+13] Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. Sublinear bounds for randomized leader election. In *Distributed Computing and Networking*, volume 7730 of *Lecture Notes in Computer Science*, pages 348–362. Springer Berlin Heidelberg, 2013.

[KY86] Anna Karlin and Andrew Chi-Chih Yao. Probabilistic lower bounds for byzantine agreement. *Manuscript*, 1986.

[LSP82a] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[LSP82b] Leslie Lamport, Robert Shostak, and Michael Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[LW11a] Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing. *CoRR*, abs/1102.5425, 2011.

[LW11b] Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing: Extended abstract. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, STOC '11, pages 11–20, San Jose, CA, USA, 2011.

[Mit01] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, Oct 2001.

[NT88] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, PODC '88, pages 248–262, New York, NY, USA, 1988. ACM.

[OBG08] Michael Okun, Amnon Barak, and Eli Gafni. Renaming in synchronous message passing systems with byzantine failures. *Distributed Computing*, 20:403–413, 2008.

[Oku10] Michael Okun. Strong order-preserving renaming in the synchronous message passing model. *Theoretical Computer Science*, 411(40-42):3787 – 3794, 2010.

[PR05] Rafael Pass and Alon Rosen. Concurrent non-malleable commitments. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '05, pages 563–572, Washington, DC, USA, 2005. IEEE Computer Society.

List of Publications

The results desribed in this thesis have been published and presented by the author in the following venues:

- Dan Alistarh, Oksana Denysyuk, Luís Rodrigues, and Nir Shavit. Balls-into-leaves: sub-logarithmic renaming in synchronous message-passing systems. *In Proceedings of the 2014 ACM symposium on Principles of distributed computing* (PODC '14). pp 232-241, 2014. ACM (invited for PODC '14 special issue in Distributed Computing journal.)
- Oksana Denysyuk and Luís Rodrigues. Byzantine renaming in synchronous systems with t < N. *In Proceedings of the 2013 ACM symposium on Principles of distributed computing* (PODC '13). pp. 210-219. ACM
- Oksana Denysyuk, Luís Rodrigues, Order-Preserving Renaming in Synchronous Systems with Byzantine Faults, *In Proceeding of the 2013 IEEE 33rd International Conference on Distributed Computing Systems* (ICDCS '13), pp. 276-285, 2013 , 2013. IEEE
- Oksana Denysyuk and Luís Rodrigues. Brief announcement: order-preserving renaming in synchronous message passing systems with byzantine faults. *In Proceedings of the 2012 ACM symposium on Principles of distributed computing* (PODC '12). pp. 233-234. ACM