UNIVERSITY OF LIVERPOOL

Performing tasks and reaching agreement in distributed systems prone to adversarial crash-failures

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Jarosław Mirek

October 2019

Dedicated to Pola

Abstract

This dissertation tackles several questions in distributed computing and fault-tolerance. It consists of four main, more or less independent, chapters that all fit into a certain bigger picture – the problem of how distributed algorithms work, when they are executed in the presence of an adversary causing specific distractions. We study fault-tolerance for two fundamental problems in distributed computing: the *Do-All* and the *Consensus* problems.

In the Do-All problem we expect t tasks to be performed by p processors in a reliable way – this means that we demand all the tasks to be performed, as a result of an algorithm execution, even though processors may be crashed by an adversary. The system is synchronized with a global clock and processors communicate via a *multiple-access channel* (MAC), which restricts simultaneous transmissions. We use *work* as a performance measure, that accrues the number of available processors steps throughout the execution of an algorithm.

Within this framework, we analyze the Do-All problem on a MAC against different, novel adversarial scenarios. In particular, we define a class of *Ordered* adaptive adversaries, which cause crashes online according to some partial order of the participating processors, that is fixed by the adversary before the execution. Furthermore, we consider a class of *Round-Delay* adaptive adversaries, who see random choices of parties with a delay.

In the second and third part of this thesis, we investigate the Do-All problem with extended assumptions about tasks. In the former part, we analyze the complexity of algorithms performing arbitrary length tasks, depending on whether *preemption* is available or not. By the means of preemption, we understand that performing tasks can be abandoned partway through without losing progress. The latter part consists of solutions for the Do-All problem with dependent tasks being in a partial order relation, where some tasks need to be performed before others.

Finally, we consider the Consensus problem in which n processes must agree on a common value. We examine the problem in a synchronous message-passing model against *Constrained* adversaries, resembling Ordered adaptive adversaries. While commonly used *Strongly-Adaptive* adversaries model malicious attacks and *Non-Adaptive* ones — faults that occur independently of the parties executing the algorithm, Constrained adversaries model more realistic scenarios with fault-prone dependent processes, e.g., in hierarchical or dependent software/hardware systems. Results from the fourth part prove that there is a separation between adversarial scenarios typically considered in the literature.

Acknowledgements

First and foremost, I thank my family for their love and support. My dad has shown me how to consequently walk my path and my mum has always been the source of warmth, resourcefulness and positive attitude. Without Mariusz, my education would not have been so successful and I would not have that sense of how to cope with different situations. Justyna is the best friend I have ever had, who bravely shares the sad and happy moments of my everyday life. What she does, against all odds, is a source of my inspiration and admiration. Pola should be the one I thank most, but the only thing I can say is that she is the brightest star I have ever seen.

I thank Prof. Darek Kowalski, for his patience and enthusiasm, without whom this dissertation would not be possible. I also thank Prof. Prudence Wong, for being a kind, understanding and always (!) helpful person, and Prof. Marek Klonowski, as he has a unique sense of humour, he treated me with great respect and taught me, perhaps, one of the most important lessons in life.

I am grateful to the Department of Computer Science of the University of Liverpool for the scholarship that made my studies possible and great facilities for doing my PhD. It was an honour to study here. I am also grateful to Dr Amitabh Trehan and Dr Othon Michail, who agreed to spend time on reviewing this thesis and acting as examiners.

Finally, I thank Tadzik Jodłowski for showing me the world of computer science and teaching honesty. I think that the way Tadzik is as a man has shaped me significantly. I am extremely happy to have met Damian Kwaśniak, Piotrek Lisowski, Krzyś Maciążek and Tomek Supel. We have had a wonderful and crazy time and it was fun to learn from one another. Without you all I would not be, where I am today.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Distributed computing is a well-established branch of computer science which studies the algorithmic aspects of performing different activities on a number of individual devices where progress and information is transferred via some communication channel. One of the problems in such a distributed environment is the so called Do-All problem.

Do-All, in its basic form, is a problem of performing t similar and independent tasks i.e., they can be performed in any sequence, in a distributed system prone to processor crashes. It was introduced by Dwork, Halpern and Waarts [41] in the context of a message-passing system. Over the years, the Do-All problem became a pillar of distributed computing and has been studied widely from different perspectives [50, 51].

The distributed system for the Do-All problem studied in this thesis is based on communication over a shared channel, also called a multiple-access channel, and was first studied in the context of the Do-All problem by Chlebus, Kowalski and Lingas [31].

The communication channel and the p crash-prone processors, also called stations, connected to it are synchronous. They have access to a global clock, which defines the same rounds for all operational stations. A message sent by station v at a round is received by all operational (i.e., not yet crashed) stations, only if v is the only transmitter in this round. We call such a transmission successful. Otherwise, unless stated differently, we assume that no station receives any meaningful feedback from the channel medium, except an acknowledgement of its successful transmission. This setting is called a channel *without collision detection.* In the setting *with collision detection* stations can distinguish between no transmission and a simultaneous transmission of at least two stations.

In this study, stations are prone to crash failures. Allowable patterns of failures are determined by abstract adversarial models. Traditionally, the main distinction was between *adaptive* and *oblivious* adversaries. The former can make decisions about failures during the computation, while the latter has to predetermine all decisions prior to the computation. Another characteristic of an adversary is its *size-boundedness*, or more specifically f-*boundedness*, if it may crash at most f stations, where $0 \leq f < p$.

In this thesis, we introduce the notion of an *Ordered-Adaptive* adversary, or simply *Ordered* adversary, which can crash stations online, but according to some preselected order (unknown to the algorithm) from a given class of partial orders. This includes linear orders, where all elements are comparable, anti-chains, where no two elements are comparable and k-thick partial orders, where at most k elements are incomparable.

Adversaries described by a partial order offer a novel framework for evaluating performance and provide better understanding of reliable distributed algorithms. For instance, in hierarchical architectures, such as clouds or networks, a crash at an upper level may result in cutting off processes at lower levels, which could be understood as a crash of lower levels processes from the system perspective. Linear orders of crashes could be motivated by the fact that each device has its own duration of operationality, unknown to the algorithm, and a crash of some device means that all with a smaller duration crashed as well. Independent systems resemble a set of linearly ordered chains of devices prone to crashes, where each chain corresponds to a different independent part of the system. Furthermore, the study in this thesis indicates that different classes of partial orders, restricting the adversary, may require different algorithmic design and yield different complexities.

Another form of restricting adversarial power is to delay the effect of its actions by a number of time steps — we call such adversaries *Round-Delayed* (*RD* for short). Such adaptive adversaries are motivated by various reactive attacking systems, especially in analyzing security aspects. We show that this parameter can influence the performance of algorithms, independently of the partial order restrictions on the adversary.

Due to the specific nature of the Do-All problem, the most accurate measure considered in the literature is *work*, aggregating the total number of available processor steps during the computation. It was introduced by Kanellakis and Shvartsman in the context of the related Write-All problem [60]. We assume that algorithms are *reliable* in the sense that they must perform all the tasks for any pattern of crashes, such that at least one station remains operational in an execution. Chlebus et al. [31] showed that $\Omega(t + p\sqrt{t})$ work is inevitable for any reliable, even randomized, algorithm for the channel with collision detection if no failures occur. This is the absolute lower bound on work complexity for the Do-All problem on a shared channel. It is known from [31] that this bound can be achieved by a deterministic algorithm for channels with *enhanced feedback*, such as collision detection, and therefore such enhanced-feedback channels are no longer interesting from the perspective of reliable task performance.

Another direction along which the Do-All problem on a shared channel is analyzed in this thesis concentrates around different assumptions about tasks. We consider tasks with arbitrary lengths, in the sense that performing different tasks might require a different number of computational steps of the processor, and study the impact of features (or their absence), distinguishing whether preemption is available, the use of randomization, and the severity of failures, on work performance.

The notion of preemption may be understood as the possibility of not performing a particular task in one attempt. This means that a single task may be interrupted partway through performing it and then resumed later by the same or even by another processor. Intuitively, the model without preemption is a more general configuration, yet both have subtleties, which distinguish them noticeably.

Another assumption for tasks, considered in this thesis, is that they are dependent and hence, some tasks must be performed before others. We model task dependencies as a partial order relation which has to be preserved during the execution. Partially ordered sets of our particular interest consist of sets of independent chains of arbitrary lengths and some types of trees.

In the Consensus problem in synchronous message-passing distributed systems there are n processes, out of which at most f can be crashed. Each process is initialized with a binary input value, and all processes must agree on a common value (from the input values). Formally, the following three properties need to be satisfied:

- *agreement*: no two processes decide on different values;
- *validity*: only a value among the initial ones may be decided upon;
- *termination*: each process eventually decides, unless it crashes.

In case of randomized solutions, the specification of Consensus needs to be reformulated, which can be done in various ways (cf., [9]). We consider a standard definition in which validity and agreement are required to hold for every execution, while a randomized reformulation for termination is that it needs to hold with probability 1.

Efficiency of algorithms is measured by the number of rounds (time complexity) until all non-faulty processes decide. This work focuses on *efficient randomized solutions* – time is understood in expected sense. On the other hand, we consider message complexity understood as the total number of messages sent in the system throughout the execution. We also provide the analysis of work performance per process and time complexity formulae depending on the desired probability of achieving it.

Randomization was used in Consensus algorithms for various kinds of failures specified by adversarial models, see [4, 9]. The reason for considering randomization is to overcome inherent limitations of deterministic solutions. Most surprising benefits of randomization is the solvability of Consensus in as small as constant time [36, 42, 88]. Feasibility of achieving small upper bounds on the performance of algorithms solving Consensus in a given distributed environment depends on the power of adversaries inflicting failures.

1.1 Thesis outline

This thesis consists of four main independent parts (Chapters 3, 4, 5, 6). Although each chapter can be read as a separate entity, they are all devoted to the analysis of fault-tolerant distributed algorithms for the Do-All problem and the Consensus problem.

In Chapter 2, we introduce the models and the problems considered in this thesis, as well as a few auxiliary results, used in subsequent chapters.

Chapters 3, 4 and 5 are closely related to each other. In all of these chapters, we analyze the Do-All problem in presence of different adversaries interfering with the system. Furthermore, we show how various assumptions imposed on tasks make the problem significantly different.

In Chapter 3, which is based on a journal paper [62], we examine the problem in its basic form, where we expect to perform t tasks, which are similar (each task has the same length), independent (tasks may be performed in any order) and idempotent (tasks may be performed many times, even concurrently by different processors - however, it suffices to perform each task once in order to solve the Do-All problem). We concentrate on randomized solutions against novel adversarial scenarios. In particular, we introduce a hierarchy of adaptive adversaries and study their impact on the complexity of performing tasks on a shared channel. The most important parameter of this hierarchy is the partial order, which describes adversarial crashes, hence we call such adversaries Ordered. Other parameters describing adversarial scenarios are the number of crashes f (i.e., the size boundedness of the adversary) and a delay c in the effect of the adversary's decisions.

The problem from Chapter 3 originates from an open question stated in [31], whether there is an algorithm for a channel without collision detection, which meets the lower bound $\Omega\left(t + p\sqrt{t} + p\min\left\{\frac{p}{p-f}, t\right\}\right)$ for work against the Weakly-Adaptive f-Bounded adversary. The authors analyzed the Do-All problem therein against different adversarial scenarios (in any case $0 \leq f \leq p-1$): Strongly-Adaptive f-Bounded adversary (whose only restriction is that it cannot crash more than f processors), Weakly-Adaptive f-Bounded adversary (who has to preselect f processors prone to crashes, prior to the execution), Weakly-Adaptive Linearly-Bounded adversary (i.e., a Weakly-Adaptive f-Bounded adversary for which $f = cp$, where $0 < c < 1$) and an Unbounded adversary, (that is a Strongly-Adaptive $(p-1)$-*Bounded* adversary).

In case of Ordered adversaries restricted by orders of arbitrary width $k \leq f$, understood as the size of the maximal anti-chain in the partial order, we present algorithm GruTEch against Ordered adversaries restricted by orders of width k. We show that it is efficient, by proving a lower bound for a broad class of partial orders. This also extends the result for a Weakly-Adaptive Linearly-Bounded adversary from [31] to an arbitrary number of crashes $f < p$, as a Weakly-Adaptive adversary is a special case of the Ordered adversary restricted by a single anti-chain. This solution provides an answer to an open question stated in [31]. Our results together with the results from [31] prove a separation between different classes of adversaries. The easiest to play against, apart from the oblivious ones, are the following adaptive adversaries: 1-RD adversaries (that is, adversaries whose decisions are delayed by a single round), Ordered adversaries restricted by short-width orders and Linearly-Bounded adversaries, restricted by a single chain. More demanding adversaries include Ordered adversaries restricted by orders of width k, for larger values of k, and f-bounded adversaries for f close to p. The most demanding adversaries are the Strongly-Adaptive ones, as their decisions and the way they act are least restricted.

Regarding the round delay of the adversary, the problem is hardest for the 0-RD adversary which is equivalent to the Strongly-Adaptive adversary. An interesting particularity is that if the Strongly-Adaptive adversary's decisions are delayed by at least one round, then we may design a solution with work complexity independent of the number of crashes.

Chapter 4, which is based on a conference paper [63], concentrates on the Do-All problem with tasks having arbitrary lengths and on studying the impact of features, including preemption, randomization, and the severity of failures, on the work performance. The notion of preemption may be understood as the possibility of not performing a particular task in one attempt. This means that a single task may be interrupted partway through performing it and then resumed later by the same processor or even by a different processor. In general, algorithms exploiting randomization may perform better and we investigate when randomization actually helps. Adversarial scenarios in this chapter include the Strongly-Adaptive adversary and the Non-Adaptive adversary, who, in addition to selecting the faulty subset of f processors, has to declare in which rounds crashes will occur, prior to the execution.

In Chapter 5, once again, we consider similar i.e., unit length tasks, where one unit length task may be performed in a single computational step of the processor. However, we assume that tasks are in a partial order relation and the resulting task dependencies have to be preserved during the execution. Partially ordered sets of our particular interest consist of sets of independent chains of arbitrary lengths and some types of trees.

The line of investigation on the Consensus problem, originated in [14], is considered in Chapter 6, which is based on a conference paper [68]. We analyze the Consensus problem against restricted adaptive adversaries. The motivation is that a Strongly-Adaptive adversary, typically used for the analysis of randomized Consensus algorithms, may not be very realistic in the sense that considering a Strongly-Adaptive adversary fulfills a policy of preparing for an absolutely worst case scenario, while in practice processes could be set as fault-prone in advance, before the execution of an algorithm, or may be dependent i.e., in hierarchical hardware/software systems. In this context, a Strongly-Adaptive adversary should be used to model malicious attacks, rather than realistic crash-prone systems. On the other hand, a Non-Adaptive adversary who must fix all its actions before the execution does not capture many aspects of fault-prone systems, e.g., attacks or reactive failures (occurring as an unplanned consequence of some actions of the algorithm in the system). Therefore, analysing the complexity of Consensus under such constraints gives a much better estimate on what may happen in real executions and, as we demonstrate, leads to new theoretical findings about the performance of Consensus algorithms.

We design and analyze a randomized algorithm against a Weakly-Adaptive adversary, which is time optimal due to the proved lower bound on the expected number of rounds.

We complement these results by showing how to modify the algorithm designed for the Weakly-Adaptive adversary, to work against a k-Chain-Ordered adversary, who has to arrange all processes into an order of k chains, and then has to preserve this order of crashes in the course of the execution.

Finally, we also consider the message complexity of the algorithms introduced in that chapter, as well as the expected work per process and consider time complexity formulae, depending on the desired probability of success.

1.2 Overview of related research

In this section, we present a digest of literature regarding the Do-All and Consensus problems, closely related with the settings considered in this thesis.

The authors in [41] analyzed the Do-All problem in the context of a message-passing model with processor crashes. The problem was studied in a number of follow-up papers, including [24, 25, 27, 38, 45] in the context of a message-passing model, in which every node can send a message to any subset of nodes in one round. Dwork et al. [41] analyzed task-oriented work, in which performing each task contributes a unit to the complexity, and the communication complexity defined as the number of point-to-point messages.

De Prisco, Mayer and Yung [38] were the first to use available processor steps [60] as the measure of work for Do-All solutions. They developed an algorithm with work complexity of order $\mathcal{O}(t + (f + 1)p)$ and message complexity $\mathcal{O}((f + 1)p)$. Galil, Mayer and Yung [45] improved the message complexity to $\mathcal{O}(fp^{\varepsilon} + \min\{f + 1, \log p\}p)$, for any positive ε, while maintaining the same work complexity. This was achieved as a by-product of their investigation of the Byzantine agreement with crash failures, for which they found a message-efficient solution. Chlebus, De Prisco and Shvartsman [24] studied failure models allowing restarts.

Chlebus and Kowalski [27] studied the Do-All problem where failures are controlled by the Weakly-Adaptive Linearly-Bounded adversary. They developed a randomized algorithm with expected effort (effort = work + number of messages) $\mathcal{O}(p \log^* p)$, in the case where $p = t$, which is asymptotically smaller than the lower bound $\Omega(p \log p/ \log \log p)$ on work of any deterministic algorithm. Chlebus, Gąsieniec, Kowalski and Shvartsman [25] developed a deterministic algorithm with effort $\mathcal{O}(t + p^a)$, for some specific constant a, where $1 < a < 2$, against the Unbounded adversary, which is the first algorithm with the property that both work and communication are $o(t + p^2)$ against this adversary. They also gave an algorithm achieving both work and communication $\mathcal{O}(t+p \log^2 p)$ against a Strongly-Adaptive Linearly-Bounded adversary. All the previously known deterministic algorithms had either work or communication performance $\Omega(t + p^2)$ when as many as a linear fraction of processing units could be failed by a Strongly-Adaptive adversary. Georgiou, Kowalski and Shvartsman [49] developed an algorithm with work $\mathcal{O}(t + p^{1+\varepsilon})$, for any fixed constant ε, by an approach based on gossiping. Kowalski and Shvartsman in [70] studied Do-All in an asynchronous message-passing mode when executions are restricted in a way that every message delay is at most d. They showed a lower bound $\Omega(t + pd \log_d p)$ on expected work. They developed several algorithms and among them a deterministic one with work $\mathcal{O}((t + pd)\log p)$. Further developments may be found in the book by Georgiou and Shvartsman [51].

Chlebus, Kowalski and Lingas [31] were the first who considered Do-All in a multiple-access channel. They showed an absolute lower bound for work complexity and a deterministic algorithm matching this performance in case of a channel with collision detection. Regarding the channel without collision detection, they developed a deterministic solution that is optimal with respect to the lower bound they proved $\Omega(t + p\sqrt{t} + p \min\{f, t\})$. This lower bound also holds for randomized algorithms against the Strongly-Adaptive adversary, (that is the adversary who can see random choices and react online), which shows that randomization does not help against a Strongly-Adaptive adversary.

Furthermore, their paper contains a randomized solution that is efficient against a Weakly-Adaptive Linearly-Bounded adversary who can only fail a constant fraction of stations. This algorithm matches the absolute lower bound on work. If the adversary is not linearly bounded, that is $f < p$, they only proved a lower bound of $\Omega(t + p\sqrt{t} + p \min\left\{\frac{p}{p-f}, t\right\})$.

Clementi, Monti and Silvestri [36] investigated Do-All in the communication model of a multiple-access channel without collision detection. They studied *F-reliable protocols*, which are correct if the number of crashes is at most F, for a parameter $F < p$. They obtained tight bounds on the time and work of F-reliable deterministic protocols. In particular, the bound on work shown in [36] is $\Theta(t + F \cdot \min\{t, F\})$. Results shown in [36] also referred to the time perspective with a lower bound on time complexity equal to

$$\Omega\left(\frac{t}{p-F} + \min\left\{\frac{tF}{p}, F + \sqrt{t}\right\}\right).$$

However, these protocols make explicit use of the knowledge of F, opposed to results from [31], as well as results presented in this thesis, hence we denote factors describing faults from [36] with a capital letter, to make this distinction visible.

Most of work in the shared channel model focused on communication problems, cf., the surveys [23, 46]. Among the most popular protocols for resolving contention on the channel are Aloha [1] and exponential backoff [84]. The two most related research problems are as follows.

The *selection problem* is about how to have an input message transmitted successfully, if only a subset of stations hold input messages, while the others do not. It is somehow closely related to the leader election problem. Willard [93] developed protocols solving the problem in expected time $\mathcal{O}(\log\log n)$ for the channel with collision detection. Kushilevitz and Mansour [72] showed a lower bound $\Omega(\log n)$ for this problem in case of a lack of collision detection, what explains the exponential gap between this model and the one with collision detection. Martel [82] studied the related problem of finding maximum within the values stored by a group of stations.

Jurdziński, Kutyłowski and Zatopiański [59] considered the leader election problem for the channel without collision detection, giving a deterministic algorithm with sub-logarithmic energy cost. They also proved a log-logarithmic lower bound for the problem.

The *contention resolution* problem is about a subset of k among all n stations which have messages. All these messages need to be transmitted successfully on the channel, as quickly as possible. Komlós and Greenberg [66] proposed a deterministic solution allowing to achieve this in time $\mathcal{O}(k + k\log(n/k))$, where n and k are known. Kowalski [67] gave an explicit solution with complexity $\mathcal{O}(k\text{polylog}n)$, while the lower bound $\Omega(k(\log n)/(\log k))$ was shown by Greenberg and Winograd [54]. The work by Chlebus, Gołąb and Kowalski [26] regarded broadcasting spanning forests on a multiple-access channel, with locally stored edges of an input graph.

A significant part of recent results for the shared channel communication model is focused on jamming-resistant protocols motivated by applications in single-hop wireless networks. To the best of our knowledge, this line of research was initiated in [12] by Awerbuch, Richa and Scheideler, wherein the authors introduced a model of adversary capable of jamming up to a $(1-\epsilon)$ fraction of the time steps (slots). The following papers [89], [90] by Richa, Scheideler, Schmid and Zhang proposed several algorithms that can reinforce the communication even for a strong, adaptive adversary. For the same model, Klonowski and Pająk in [65] proposed an optimal leader election protocol, using a different algorithmic approach.

The similar model of a jamming adversary was considered by Bender, Fineman, Gilbert and Young in [16]. The authors consider a modified, robust exponential backoff protocol that requires $O(\log^2 n + T)$ attempts in the channel, if there are at most T jammed slots. Motivated by saving energy, the authors try to find maximal global throughput while reducing device costs expressed by the number of attendants in the channel.

Finally, there are several recent results on finding approximations of the network. In [19] Brandes, Kardas, Klonowski, Pająk and Wattenhofer proposed an algorithm for the network of n stations that returns $(1+\varepsilon)$ - approximation of n with probability at least $1 - 1/f$.

This procedure takes $O(\log\log n + \log f/\varepsilon^2)$ time slots and was also proved to be time-optimal. In [22] Chen, Zhou and Yu demonstrated a size approximation protocol for a seemingly different model (namely a RFID system) that needs $\Omega(\frac{1}{\epsilon^2 \log 1/\epsilon} + \log\log n)$ slots for $\epsilon \in [1/\sqrt{n}, 0.5]$ and has a negligible probability of failure. In fact, this result can be instantly translated into the MAC model.

The investigation undertaken in this thesis is motivated by two different streams of research and combines the classic scheduling problem of performing tasks with the problem of performing tasks considered in distributed computing.

Scheduling with precedence constraints was studied on different objectives for decades [73], Lenstra and Rinnooy Kan [76], Lawler [74]. these investigations had been initiated on a single machine (Lawler [73], Lawler [74], Lenstra and Rinnooy Kan [76], Leung and Young [78], Chekuri and Motwani [21], Woeginger [94]) and afterwards extended to multiple machines (Lawler [75], Lushchakova [81], Hurink and Knust [58], Skutella and Uetz [92]). The problem was studied in more complex settings, under various assumptions, e.g., preemption was considered by Lawler [75] and Lushchakova [81]. Recently, precedence constraints were studied on machines with different speeds by Chudak and Shmoys [35] and variable speeds (Pruhs, van Stee and Uthaisombut [87], Bampis, Letsios and Lucarelli [13]). More details on this line of research can be found in Leung [77].

Extending the study of unit length tasks to arbitrary length tasks took place in [71], where one can find recent results and references to such extensions in the context of fault-tolerant centralized scheduling. There has been a long history of studying the preemptive vs non-preemptive model, e.g., [83, 80, 37], to which results of this thesis also contribute.

Consensus is one of the fundamental problems in distributed computing, with a rich history of research done in various settings and systems, cf., [9]. Recently its popularity grew even further due to applications in emerging technologies such as blockchains.

Consensus is solvable in synchronous systems with processes prone to crashing, although time $f+1$ is required [43] and sufficient [47] in case of deterministic solutions, where f represents the number of failures. Chor, Meritt and Shmoys [34] showed that randomization allows to obtain a constant expected time algorithm against a Non-Adaptive adversary, if the minority of processes may crash.

Bar-Joseph and Ben-Or [14] proved a lower bound $\Omega(f/\sqrt{n\log n})$ on the expected time for randomized Consensus against the Strongly-Adaptive adversary and proposed an algorithm reaching Consensus in $\mathcal{O}(f/\sqrt{n\log(2+f/\sqrt{n})})$ for any $f < n$. This solution meets their lower bound, provided that the adversary can fail $f = \Omega(n)$ processes. What is more, for such condition these bounds reformulate to $\Theta(\sqrt{n/(n \log n)})$.

Fisher, Lynch and Paterson in a landmark work [44] showed that for the message-passing model Consensus cannot be solved deterministically in *asynchronous settings*, even if only one process may crash. Loui and Abu-Amara [79] showed a corresponding result for shared memory. These impossibility results can be circumvented when randomization is used and the Consensus termination condition does not hold with some small probability.

Bracha and Toueg [18] observed that it is impossible to reach Consensus by a randomized algorithm in the asynchronous model with crashes if the majority of processes are allowed to crash. Ben-Or [15] gave the first randomized algorithm solving Consensus in the asynchronous message-passing model under the assumption that the majority of processes are non-faulty.

For the strong adversary Abrahamson [1] gave the first randomized solution in the shared memory model, which had exponential work, and Aspnes and Herlihy [6] gave the first polynomial-work algorithm for the same adversary. Aspnes and Waarts [7], Bracha and Rachman [17], and Aspnes, Attiya and Censor [5] gave solutions with $\mathcal{O}(n \,\text{polylog} n)$ work per process. Aspnes [3] showed that this was best possible up to a poly-logarithmic factor by showing that any algorithm for the strong adversary requires $\Omega(f^2/\log^2 f)$ work by proving that these many coin flips were needed. Attiya and Censor [8] resolved the work complexity of randomized Consensus in shared memory against the strong adversary by showing that $\Theta(n^2)$ total work is both sufficient and necessary, building on the work of Aspnes and Herlihy [6], Bar-Joseph and Ben-Or [14], and Moses and Rajsbaum [86]. Research on weak adversaries [10, 11, 20], that may observe coin flips and the content of shared registers, resulted in scalable algorithms of $\mathcal{O}(n \,\text{polylog} n)$ total work and $\mathcal{O}(\text{polylog} n)$ work per process.

The Consensus problem has been recently considered against different adversarial scenarios. Robinson, Scheideler and Setzer [91] considered the synchronous Consensus problem under a *late ϵ-bounded adaptive* adversary, whose observation of the system is delayed by one round and can block up to ϵn nodes in the sense that they cannot receive and send messages in a particular round.

Message complexity of Consensus was extensively studied before. Amdur, Weber and Hadzilacos [2] showed that $\Omega(n)$ messages need to be sent by a protocol solving Consensus in an eventually synchronous environment in a failure-free execution. Dolev and Reischuk [40] and Hadzilacos and Halpern [55] proved the $\Omega(fn)$ lower bound on the message complexity of deterministic Consensus for Byzantine failures.

Chlebus and Kowalski [30] showed that Consensus can be solved with the aid of a randomized algorithm that is locally scalable with respect to both time and bit communication complexities, when the number f of failures is at most a constant fraction of the number n of processes and the adversary is Non-Adaptive. A comparable performance is impossible to achieve by a deterministic solution, as was observed in [32]. A deterministic algorithm solving Consensus in the synchronous setting that has processes sending a total of $\mathcal{O}(n \,\text{polylog} n)$ bits per message, meaning the algorithm is globally scalable, and which can handle up to $n-1$ crashes, was developed in [32]. The globally scalable solution given in [32] is deterministic but it is not explicit.

Gilbert and Kowalski [53] presented a randomized Consensus algorithm that achieves optimal communication complexity, using $\mathcal{O}(n)$ bits of communication and terminates in $\mathcal{O}(\log n)$ time with high probability, tolerating up to $f < n/2$ crash failures against a Non-Adaptive adversary.

Deterministic consensus solutions operating in time $\mathcal{O}(f)$ with sub-quadratic message complexities usually rely on gossiping procedures, as given, for instance, in [28, 29, 52]. All such algorithms send $\Omega(n^2)$ bits in messages in the worst case, due to the nature of gossiping. Solutions given in [29, 52] are *early stopping* i.e., they terminate in $\mathcal{O}(f)$ rounds, where $f \leq n$ is the number of failures that actually occur in a given execution. Georgiou, Gilbert, Guerraoui and Kowalski [48] gave a randomized solution for asynchronous message-passing systems with processes prone to crashes, which sends $o(n^2)$ messages, but their algorithm has $\omega(n^2)$ bit complexity.

Deterministic Byzantine consensus requires $\Omega(fn)$ message complexity, which can be improved with the use of randomization. A Byzantine consensus algorithm achieves *consensus with loss* $h(f)$, if at least $n - h(f)$ non-faulty processes eventually decide on a common value, when up to f processes are corrupted. An *almost-everywhere consensus* means consensus with loss $h(f)$, for some function $h(f) \leq cf$, where $c < 1$ is a constant. The first randomized solutions for almost-everywhere Byzantine consensus and leader election scalable with respect to bit communication complexity were given by King, Saia, Sanwalani and Vee [61]. A trade-off between the time required to solve an almost everywhere Byzantine consensus by a randomized scalable solution and the magnitude of loss was proved by Holtby, Kapron and King [57]. They showed that if f is a constant fraction of n, then a scalable solution requires $\Omega(n^{1/3})$ rounds.

1.3 Author's publications

Results presented in this thesis are the author's and his coauthors' novel contribution to the field. All of them constitute this dissertation, with an exception of Section 3.1, results from which were the basis of the author's master's thesis: "The Do-All problem and its extensions", defended at the Wrocław University of Science and Technology in academic year 2015/2016.

Title	**Authors**	**Venue**
Ordered and delayed adversaries and how to work against them on a shared channel [62]	M. Klonowski, D.R. Kowalski J. Mirek	Distributed Computing (2018)
Fault-Tolerant Parallel Scheduling of Arbitrary Length Jobs on a Shared Channel [63]	M. Klonowski, D.R. Kowalski J. Mirek, P.W.H. Wong	FCT 2019 [1]
Performing Partially Ordered Sets of Jobs on a MAC in Presence of Adversarial Crashes [64]	M. Klonowski, D.R. Kowalski J. Mirek, P.W.H. Wong	NCA 2019 [2]
On the Complexity of Fault-Tolerant Consensus [68]	D.R. Kowalski, J. Mirek	NETYS 2019 [3]

Table 1.1: Author's publications on which the thesis is based.

Results from this thesis contribute twofold to the state-of-the-art. On one hand, we introduce novel adversarial scenarios, which we refer to as Ordered adversaries (for the Do-All problem) and Constrained adversaries (for the Consensus problem). They offer a more realistic framework for evaluating performance of distributed algorithms. To the best of our knowledge, adversaries, whose patterns of crashes are described by a partial order were not considered in the literature so far. Their properties prove, that the difficulty of the problems grows with the size of the maximal anti-chain, describing the adversary.

On the other hand, we consider the Do-All problem on a multiple-access channel with extended assumptions about tasks. This includes tasks with arbitrary lengths and tasks which have dependencies between each other. These results make a contribution to the scheduling theory on a multiple-access channel, which was not considered deeply before and may elevate an interesting line of research for future work.

[1] 22nd Symposium on Fundamentals of Computation Theory

[2] 18th IEEE International Symposium on Network Computing and Applications

[3] 7th Edition of The International Conference on NETworked sYStems

Chapter 2

Preliminaries

In this chapter, we introduce the formal model for the problems considered in this thesis. The first section is devoted to the model for the Do-All problem. It forms the basis for considerations in Chapters 3, 4 and 5. Additionally, in Sections 2.1.5, 2.1.6 and 2.1.7 we provide different assumptions on tasks, which distinguish the problems analyzed in the corresponding chapters.

In Section 2.2, we provide appropriate details for Chapter 6, where we consider the Consensus problem in a synchronous message-passing distributed system.

Section 2.3 describes fundamental algorithmic building blocks, which we use throughout this thesis.

2.1 Model for the Do-All problem

The Do-All problem has been introduced by Dwork et al [41] and was investigated further in numerous papers [33, 25, 27, 38, 45] under different assumptions regarding the model. In this section, we formulate the model for our considerations, which is based on the one presented in [31].

In general, the Do-All problem is modeled as follows: a distributed system of computationally constrained devices is expected to perform a number of tasks [50]. We will call those devices *processors* or simply *stations*. The main efficiency measure that we use is *work*, i.e., the total number of processor steps available for computations [60].

2.1.1 Stations

In our model we assume having p stations (which we also refer to as processors or devices), with unique identifiers from the set $\{1, \ldots, p\}$, without loss of generality. The distributed system of those devices is synchronized by a global clock, and time is divided into synchronous time slots, called *rounds*. All the stations start simultaneously at a certain moment. Furthermore, every station may halt voluntarily. In this thesis, by n we will denote the number of operational, i.e., not crashed, stations.

2.1.2 Communication

The primary communication medium for processors is a multiple-access channel e.g., [23, 46], where a single message, transmitted in a particular round, reaches every operational device. All our solutions work on a channel *without collision detection*, hence when more than one message is transmitted at a certain round, then devices hear a signal indistinguishable from the background noise.

In few places of this work, we refer to the alternative channel setting *with collision detection*, in which there are three types of feedback from the channel:

- **Silence** — no station transmits, and only a background noise is heard;
- **Single** — exactly one station transmits a legible information;
- **Collision** — an illegible signal is heard (yet different from *Silence*), when more than one station transmits simultaneously.

Section 2.3.2 provides more details regarding how collision detection may be utilized from the algorithmic perspective. Let us note, that the setting with collision detection is referred to only in the context of transforming algorithmic tools and lower bounds to the more challenging setting without collision detection primarily studied in this work.

In Section 3.5 of this thesis, we extend one of the results to the beeping model. In the beeping model we distinguish two types of signals. One is silence, where no station transmits. The other is a beep, which, when heard, indicates that at least one station transmitted.

2.1.3 Adversaries

Processors may be crashed by the adversary. One of the factors that describes the adversary is its f-boundedness. It represents the total number of failures that may be enforced. We assume that $0 \leq f \leq p-1$, so always at least one station remains operational until an algorithm terminates. Stations that were crashed neither restart nor contribute to work. What is more, crashes can take place at any point of a particular round. Another feature of the adversary is whether it is adaptive or not. Following the definition from [50], an *adaptive adversary* is the one that has complete knowledge of the computation that it is affecting, and it makes instant dynamic decisions on how to affect the computation. In particular, this means that adaptive adversaries can decide to crash processors at any time of the algorithm execution. What is more A *non-adaptive adversary*, sometimes called *oblivious*, has to determine a sequence of events it will cause before the start of the computation. In chapters dedicated to the Do-All problem we focus on adaptive adversaries. All adversaries we consider in this thesis are f-Bounded. Hence, throughout the thesis e.g., instead of writing Strongly-Adaptive f-Bounded adversary, we simply write Strongly-Adaptive adversary.

In the relevant literature, the following adversarial models were considered, cf., [31],[51]:

- **Strongly-Adaptive f-Bounded**: the only restriction of this adversary is that the total number of failures may not exceed f. In particular all possible failures may happen simultaneously.
- **Weakly-Adaptive f-Bounded**: the adversary has to declare a subset of f stations prone to crashes before the algorithm execution. Yet, it may arbitrarily perform crashes on the declared subset.
- **Unbounded**: that is Strongly-Adaptive $(p-1)$-Bounded.
- **Linearly-Bounded**: an adversary of power f, where $f = cp$, for some $0 < c < 1$.

We now introduce new adversarial models that complement existing ones in literature.

The Ordered f-Bounded adversary

Formally, the Ordered f-Bounded adversary has to declare, prior to the execution, a subset of at most f out of p stations that will be prone to crashes. Then, before starting the execution, the adversary has to determine a partial order on the selected stations, taken from a given family of partial orders. This family restricts the power of the adversary — the wider it is the more power the adversary possesses. Moreover, as we will show in this work, the structure of available partial orders influences the asymptotic performance of algorithms and the complexity of the Do-All problem in presence of an adversary restricted by these partial orders.

The adversary may enforce a failure independently of time slots (even f at the same round), but without violating the order. This means that a pre-selected crash-prone station can be crashed in a time slot if and only if all stations preceding it in the order have been already crashed by the end of that time slot.

In this work we focus on the following three families of partial orders.

- Linearly-Ordered f-Bounded: formally, the Linearly-Ordered f-Bounded adversary has to choose a sequence $\pi = \pi(1) \ldots \pi(f)$ designating the order on the selected set of f stations in which failures will occur, where $\pi(i)$ represents the identifier of the ith fault-prone station in the order. This means that station $\pi(i)$ may be crashed if and only if stations $\pi(j)$ are already crashed, for all $j < i$. Thus, the notion of sequence π, which represents a linear partial order.

- k-Chain-Ordered f-Bounded and k-Thick-Ordered f-Bounded: the k-Chain-Ordered f-Bounded adversary has to arrange f pre-selected stations into a partial order consisting of k disjoint chains of arbitrary lengths, which represent in what order these stations may be crashed. This is equivalent to k disjoint linear orders, as described for the Linearly-Ordered f-Bounded adversary.

  We denote l_j, $1 \leq j \leq k$, as the length of chain j and we assume that the sum of lengths of all chains equals f.

  While considering k-Chain-Ordered adversaries we will also define additional notions, useful in the analysis of certain results. We say that a partial order is a *k-chain-based partial order* if it consists of k disjoint chains such that:

  - no two of them have a common successor, and
  - the total length of the chains is a constant fraction of all elements in the order.

  Furthermore, by the *thickness* of a partial order P we understand the maximal size of an anti-chain in P. An adversary restricted by a wider class of partial orders of thickness k is called a k-Thick-Ordered f-Bounded adversary.

- Anti-Chain-Ordered f-Bounded it is restricted by a partial order which is the anti-chain of f elements, i.e., all f crash-prone stations are incomparable, thus could be crashed in any order. This adversary is consistent with the Weakly-Adaptive f-Bounded adversary and the k-Thick-Ordered f-Bounded ones.

The c-Round-Delay f-Bounded adversary

The c-Round-Delay f-Bounded adversary's (c-*RD* for short) decisions take effect with a c round delay. This means that if the adversary decides to interfere with the system by crashing a processor, then this will inevitably happen after c rounds. In particular, this means that the subsequent execution and random bits do not influence the decision and its effect — the decision is final and once made by the adversary cannot be changed during the delay. We still consider f-boundedness of the adversary, but apart from that, it may decide arbitrarily, without declaring which stations will be prone to crashes before the algorithm execution.

Special cases of the c-RD adversary of our particular interest are a 0-*RD* and a 1-*RD* adversary model. The definition of the former model is consistent with the Strongly-Adaptive adversary, which was studied before. In this thesis, we analyze the latter model, which gives an answer to the question of how a delay influences the difficulty of the problem for a strong adversary.

The Non-Adaptive f-Bounded adversary

A Non-Adaptive f-Bounded adversary, in addition to choosing the faulty subset of f machines, has to declare, in the beginning of the execution, in which rounds crashes will occur, i.e., for every processor declared as faulty, there must be a corresponding round number in which the fault will take place.

2.1.4 Main complexity measure

The complexity measure that is mainly used in our analysis is *work*, as mentioned before. It is the number of available processor steps for computations. This means that each operational processor contributes a unit of work in each round, as long as it did not halt, even if it is idling.

More formally, let us consider algorithm A and assume that execution $\mathcal{E}$ starts when all the processors begin simultaneously in some fixed round r_0. Let r_v be the round when processor v halts or is crashed. Then its work contribution is equal to $r_v - r_0$. In what follows, work accrued by A in execution $\mathcal{E}$ is the sum of such expressions over all processors, i.e., $\sum_{1 \leq v \leq p}(r_v - r_0)$. The work complexity of A is the maximum over all possible executions of A.

For randomized algorithms, we assess the *expected work*, which is defined as the maximum over all executions of the expectation of the sum above.

Work is a standard complexity measure for analyzing the Do-All problem in various models, details to be found in [51]. It is worth emphasizing that time, as the most customary complexity measure, does not play a central role in the analysis of task-performing algorithms in the presence of an adversary. Opposed to work, time does not capture certain subtleties and often miscalculates the actual effort of the algorithm.

2.1.5 Tasks in Chapter 3

We assume that tasks are *similar* (that is, each task requires the same number of rounds to be carried out), *independent* (they can be performed in any order) and *idempotent* (every task may be performed many times, even concurrently by different processors without affecting the outcome of its computation). We assume that one round is sufficient to perform a single task.

2.1.6 Tasks in Chapter 4

We assume that *tasks have arbitrary lengths*, are *independent* and *idempotent*. We assume that tasks have some minimal (atomic or unit) length. Consequently, we can also assume that each task's length is a multiple of the minimal length. As the model that we consider is synchronous, this minimal length may be justified by the round duration required for local computations for each processor. By ℓ_a we denote the length of task a. We also use L to denote the sum of lengths of all tasks, i.e., $L = \sum_i \ell_i$. Finally, by α we denote the length of the longest task.

Preemptive vs Non-Preemptive Model

By the means of *preemption* we define the possibility of performing tasks in several pieces. Let us consider task a, of length ℓ_a and let processor v be scheduled to perform task a at some point of the algorithm execution. Assume that v performs x units of task a and then reports such progress. When preemption is available the remaining $\ell_a - x$ units of task a may be performed by any processor $w \neq v$.

Length ℓ_a of task a means that task a requires ℓ_a rounds to be fully performed. Such a view allows to think that task a is a chain of ℓ_a *subtasks*. Hence, we conclude that all tasks form a set of chains of unit length subtasks.

We further denote by a_k subtask k of task a. However, when we refer to a single task, disregarding its subtasks, we refer to it simply as task a.
We define two types of tasks depending on how intermediate progress is handled:

- **Oblivious task** — it is sufficient to have knowledge that previous subtasks were completed, in order to perform remaining subtasks from the same task. In other words, any information from in-between progress does not have to be announced.

- **Non-Oblivious task** — any intermediate progress needs to be reported through the channel when interrupting the task to resume it later, and possibly pass the task to another processor.

In the preemptive oblivious model a task may be abandoned by processor v on some subtask k without confirming progress up to this point on the channel and then continued from the same subtask k by any processor w.

As an example of the preemptive oblivious model, consider a scenario where a shared array of length x needs to be erased. If a processor stopped performing this task at some point, another one may reclaim the task without the necessity of repeating previous steps, by simply reading to which point it has been erased.

For the preemptive non-oblivious model, consider that a processor executes Dijkstra's algorithm for finding the shortest path. If it becomes interrupted, then another processor cannot reclaim this task otherwise than by performing the task from the beginning, unless intermediate computations have been shared. In other words preemption is available with respect to maintaining information about tasks.

On the contrary to the preemptive model, we also consider the model without preemption i.e., where each task can be performed by a processor only in one piece - when a processor is crashed while performing such task, the whole progress is lost, even if it was reported on the channel before the crash took place.

2.1.7 Tasks in Chapter 5

Tasks are assumed to be *similar* and *idempotent*, which respectively means that they require the same number of rounds to be done and that every task can be performed many times, even concurrently by different processors. Furthermore, we assume that tasks are *dependent* i.e., the order in which they should be performed is described by a partial order relation.

The relation of our particular interest while considering partially ordered tasks is the precedence relation $\prec$. If $a \prec b$ in the partial order, then we say that task a precedes task b and in what follows task a must be done before task b. If it holds that $a \prec b$ or $b \prec a$, we say that a and b are comparable. Consequently, a subset of tasks where each pair of tasks is comparable is called a *chain*. Note that in the case of tasks ordered in a chain parallelisation of performing tasks is impossible and they have to be done one-by-one. A subset of tasks where no two different tasks are comparable is called an anti-chain.

A task is considered *done*, when it is performed by a processor. A task is *confirmed*, when a processor broadcasts the fact of it being *done*. Let us consider some task a. Before a particular processor begins working on a, all tasks preceding a must be confirmed on the channel. This assumption covers also a real-life scenario, where the confirmation of task a contains an input necessary for performing a's direct successors. Such approach is natural, for example in many problems of the MapReduce paradigm.

Performing tasks in a certain order and confirming them via the channel is motivated by the need of obtaining outputs of previous tasks as the input for the following ones. Hence, we assume that processors transmit results of certain tasks in a confirmation broadcast and this opens access to the proceeding tasks for all processors. If some processor is working on a particular chain of tasks, then it may be doing consecutive tasks in this chain without confirming them on the channel until the last task in the chain is done, as it has required inputs, for consecutive tasks, computed locally.

It is convenient to think about the partial order of tasks from the Hasse diagram perspective. The notions of chains and anti-chains seem to be intuitive when graphically presented, e.g., a chain is a pattern of consecutive tasks that have to be done while an anti-chain provides flexibility in the order in which tasks should be done.

Throughout Chapter 5, by H we denote the length of the longest chain of tasks.

2.1.8 Reliability and common assumptions about tasks

We expect that processors will perform all t tasks as a result of executing an algorithm. We assume that processors have entire knowledge about tasks i.e., their identifiers, lengths and, possibly, what are the dependencies between them.

We assume that all our algorithms need to be reliable. A *reliable* algorithm satisfies the following conditions in any execution: *all the tasks are eventually performed, if at least one station remains non-faulty* **and** *each station eventually halts, unless it has crashed.*

2.1.9 Do-All formal definition

Having explained assumptions for tasks, we may now state the formal definition of the Do-All problem after [50], specifically for the multiple-access channel model:

Do-All: Given a set of t tasks, perform all tasks reliably, using p processors, under adversary $\mathcal{A}$, where p and t are known to processors and communication takes place via a multiple-access channel.

In our considerations, adversary $\mathcal{A}$ from the definition above is one of the adversaries described in Section 2.1.3.

The Do-All problem may be considered completed or solved. It is considered *completed* when all tasks are performed, but their outcomes are not necessarily known by all operational stations. The problem is considered *solved* if in addition all operational processors are aware of the tasks' outcomes. In this thesis we do not assume that stations need to know tasks' outcomes, yet all our algorithms are designed in such a way that they may solve the problem, with an exception of algorithms GruTEch and GILET from Section 3, which complete it (performing tasks is confirmed by a collision signal, which does not contain any meaningful message).

2.1.10 Inherent cost of reliability

Having explained the essential properties of the model for the Do-All problem, let us now emphasize on where is the difficulty of the problem.

Let us consider a trivial approach to the Do-All problem, where each processor performs all t tasks. Clearly, since the adversary is f-bounded and $0 \leq f < p$, then always at least one processor remains operational and hence, after t rounds we may be sure to complete the problem. Even though such a solutions is reliable, it generates pt work.

On the other hand, let us consider a different approach, where we halt some processors, in order to prevent generating excessive work. Without loss of generality, let us assume that we halt a single processor v at the beginning of the algorithm execution. Then, the Unbounded adversary, who can crash up to $p - 1$ processors, crashes all processors but v, which cannot restart, so we end up with undone tasks. Since reliability is a crucial assumption, we conclude that arbitrary halts are excluded in our model.

The difficulty of the problem is, therefore, how to utilize all processors and overcome the limitations of the multiple-access channel efficiently, in order to minimize overall work.

Reliability does not allow a station to halt, if there are still some remaining tasks. Therefore, there is a certain amount of work to be accrued, while solving the Do-All problem, even in an optimistic scenario with few crashes. We end this section with a lemma from [31] together with its proof, stating the minimal work for the considered model.

Lemma 1. *([31], Lemma 2) A reliable algorithm, possibly randomized, performs $\Omega(t + p\sqrt{t})$ work in an execution in which no failures occur.*

Proof. It is sufficient to consider the channel with collision detection, in which stations could possibly have more information. Let $\mathcal{A}$ be a reliable algorithm. The part $\Omega(t)$ of the bound follows from the fact that every task has to be performed at least once in any execution of $\mathcal{A}$.

Task α is *confirmed* at round i of an execution of algorithm $\mathcal{A}$, if either a station broadcasts successfully and it has performed α by round i, or at least two stations broadcast simultaneously and all of them, with a possible exception of one station, have performed task α by round i of the execution. At least half of the stations broadcasting at round i and confirming α have performed it by then, so at most $2i$ tasks can be confirmed at round i. Let $\mathcal{E}_1$ be an execution of the algorithm when no failures occur. Let station v come to a halt at some round j in $\mathcal{E}_1$.

Claim: Tasks not confirmed by round j were performed by v itself in $\mathcal{E}_1$.

Proof of the Claim. Suppose, to the contrary, that this is not the case, and let β be such a task. Consider an execution, say $\mathcal{E}_2$, obtained by running the algorithm and crashing any station that performed task β in $\mathcal{E}_1$ just before it was to perform β in $\mathcal{E}_1$, and all the remaining stations, except for v, crashed at step j. The broadcasts on the channel are the same during the first j rounds in $\mathcal{E}_1$ and $\mathcal{E}_2$. Hence, all the stations perform the same tasks in $\mathcal{E}_1$ and $\mathcal{E}_2$ until round j. The definition of $\mathcal{E}_2$ is consistent with the power of the Unbounded adversary. The algorithm is not reliable because task β is not performed in $\mathcal{E}_2$ and station v is operational. This justifies the claim. □

We estimate the contribution of station v to work. The total number of tasks confirmed in $\mathcal{E}_1$ is at most

$$2(1 + 2 + \ldots + j) = \mathcal{O}(j^2) \ .$$

Suppose some t' tasks have been confirmed by round j. The remaining $t - t'$ tasks have been performed by v. The work of v is at least

$$\Omega(\sqrt{t'} + (t - t')) = \Omega(\sqrt{t}) \ ,$$

which completes the proof. □

2.2 Model for the Consensus problem

2.2.1 Synchronous distributed system

We assume having a system of n processes, that communicate in the message-passing model. This means that processes form a complete graph where each edge represents a communication link between two processes. If process v wants to send a message to process w, then this message is sent via link (v, w). It is worth noticing that links are symmetric, i.e., $(v, w) = (w, v)$. We assume that messages are sent instantly.

Following the synchronous model by [14], we assume that computations are held in a synchronous manner and hence time is divided into rounds consisting of two phases:

- Phase A - generating local coins and local computation.
- Phase B - sending and receiving messages.

2.2.2 Adversarial scenarios

Processes are prone to crash-failures that are a result of the adversary activity. The adversary of our particular interest is an adaptive one - it can make arbitrary decisions and see all local computations and local coins, as well as messages intended to be sent by active processes. Therefore, it can decide to crash processes during phase B. Additionally while deciding that a certain process crashes, it can decide which subset of messages will reach their recipients, after the crash takes place.
In the context of the adversaries from Chapter 6 we distinguish three types of processes:

- Crash-prone - processes that can be crashed by the adversary.
- Fault-resistant - processes that are not in the subset of the Weakly-Adaptive$_{\mathrm{C}}$ adversary and hence cannot be crashed.

- Non-faulty - processes that survived until the end of the algorithm.

Throughout this thesis, we consider several types of adversarial scenarios for the Consensus problem:

– *Strongly-Adaptive$_C$ and Weakly-Adaptive$_C$ adversaries.* The only restriction for the Strongly-Adaptive$_{\mathrm{C}}$ adversary is that it can fail up to f processes, where $0 \leq f < n$.

  The Weakly-Adaptive adversary is restricted by the fact that before the algorithm execution it must choose f processes that will be prone to crashes, where $0 \leq f < n$.

  Observe that for deterministic algorithms the Weakly-Adaptive$_{\mathrm{C}}$ adversary is consistent with the Strongly-Adaptive$_{\mathrm{C}}$ adversary, because it could simulate the algorithm before its execution and decide on choosing the most convenient subset of processes.

– *k-Chain-Ordered$_C$ and k-Ordered$_C$ adversaries.* The notion of a k-Chain-Ordered$_{\mathrm{C}}$ adversary originates from partial order relations, hence appropriate notions and definitions translate straightforwardly. We consider the precedence relation. If some process v precedes process w or w precedes v in the partial order of the adversary, then we say that v and w are comparable. This means that either process v must be crashed by the adversary before process w or w must be crashed before v, accordingly.

  Formally, the k-Chain-Ordered$_{\mathrm{C}}$ adversary has to arrange **all** the processes into a partial order consisting of k disjoint chains of arbitrary length that represent in what order these processes may be crashed.

  By the *thickness* of a partial order P we understand the maximal size of an anti-chain in P. An adversary restricted by a wider class of partial orders of thickness k is called a k-Ordered$_{\mathrm{C}}$ adversary.

  We refer to a wider class of adversaries for the Consensus problem, constrained by an arbitrary partial order, as Ordered$_{\mathrm{C}}$ adversaries. What is more, adversaries having additional limitations, apart from the possible number of crashes (i.e. all described in this section, but the Strongly-Adaptive$_{\mathrm{C}}$ adversary), will be called Constrained adversaries. Note that Ordered$_{\mathrm{C}}$ adversaries are also restricted by the number of possible crashes f they may enforce. Additionally, throughout Chapter 6 we denote l_j as the length of chain j.

– *Non-Adaptive$_C$ adversaries.* The Non-Adaptive$_{\mathrm{C}}$ adversaries are characterised by the fact that they must fix all their decisions prior to the execution of the algorithm and then follow this pattern during the execution.

Note that the definitions of Strongly-Adaptive$_{\mathrm{C}}$ and Weakly-Adaptive$_{\mathrm{C}}$ adversaries are consistent with the corresponding ones for the Do-All problem. However, they differ by the fact that the consensus adversaries can decide which subset of messages will reach their recipients, after the crash takes place. On the other hand, the definition of a k-Chain-Ordered$_{\mathrm{C}}$ adversary differs from the k-Chain-Ordered f-Bounded adversary from Section 2.1 by the fact that the k-Chain-Ordered$_{\mathrm{C}}$ adversary has to arrange all processes into chains and cannot exceed f crashes, while the k-Chain-Ordered f-Bounded adversary chooses a faulty subset and orders these f processes into chains. Hence, in order to avoid any ambiguities, all adversaries from Chapter 6 have a $_C$ subscript.

2.2.3 Consensus formal definition

In the Consensus problem n processes, each having its input bit $x_i \in \{0, 1\}$, $i \in \{1, \ldots, n\}$, have to agree on a common output bit in presence of the adversary, capable of crashing processes. We require any Consensus protocol to satisfy the following conditions:

- Agreement: all non-faulty processes decide the same value.

- Validity: if all processes have the same initial value x, then x is the decision value.

- Termination: all non-faulty processes decide with probability 1.

2.2.4 Complexity measures

The main complexity measure used to benchmark the Consensus problem is the number of rounds by which all non-faulty processes decide. On the other hand we also discuss message complexity, understood as the total number of messages sent in the system, until all non-faulty stations decide on a common value.

We also analyze expected work per process understood as the expected number of rounds of a single process during the execution, when it stays non-idle.

2.3 Algorithmic building blocks

In this section, we present algorithms that we use in order to clarify the presentation throughout the thesis. We use them as given, yet here we provide appropriate details.

2.3.1 Algorithm Two-Lists

In this subsection we describe a deterministic *Two-Lists* algorithm from [31] which is used in our solutions as a sub-procedure. It was proved that this algorithm is asymptotically optimal for the Weakly-Adaptive adversary on a channel without collision detection, and its work complexity is $\mathcal{O}(t + p\sqrt{t} + p\ \min\{f, t\})$. The characteristic feature of Two-Lists is that its complexity is linear for some ranges of the p and t parameters, describing the number of processors and tasks, respectively (for details, see Fact 8 in Section 3.1).

Algorithm 1: Two-Lists, pseudo-code for processor v; from [31]

```
initialize STATIONS to a sorted list of all $p$ names of stations;
initialize both TASKS and OUTSTANDING$_v$ to sorted list of all $t$ names of tasks;
initialize DONE$_v$ to an empty list of tasks;
repeat
    Epoch-Two-Lists;
until halted;
```

Basic facts and notation

Two-Lists was designed for a multiple-access channel without collision detection. That is why simultaneous transmissions are excluded therein. It has been realized by a cyclic schedule of broadcasts (round-robin). This means that stations maintain a transmission schedule and broadcast one by one, accordingly. Because of this design every message transmitted via the channel is legible to all operational stations.

Another important fact about Two-Lists is that stations maintain the list of tasks, which enables them to distinguish which tasks they are responsible for. Both the tasks list and the transmission schedule are maintained as common knowledge. The result of such an approach is that stations may transmit messages of a minimal length, just to confirm that they are still operational and performing their assigned tasks.

Additionally, the transmission schedule and tasks list is stored locally on each station,

Algorithm 2: Epoch-Two-Lists, pseudo-code for processor v; from [31]

```
set pointer Task_To_Do$_v$ on list TASKS to the initial position of the range $v$;
set pointer Transmit to the first item on list STATIONS;
repeat
    // Round 1:
    perform the first task on list TASKS, starting from the one pointed to by
    Task_To_Do$_v$, that is, in list OUTSTANDING$_v$ move the performed task from list
    OUTSTANDING$_v$ to list DONE$_v$;
    advance pointer Task_To_Do$_v$ by one position on list TASKS;
    // Round 2:
    if Transmit points to $v$ then
        broadcast one bit;
    end
    attempt to receive a message;
    // Round 3:
    if a broadcast was heard in the preceding round then
        for each item $x$ on list DONE$_{Transmit}$ do
            if $x$ is on list OUTSTANDING$_v$ then
                move $x$ from OUTSTANDING$_v$ to DONE$_v$;
            end
            if $x$ is on list TASKS then
                remove $x$ from TASKS;
            end
        end
        if list TASKS is empty then
            halt;
        end
        advance pointer Transmit by one position on list STATIONS;
    end
    else
        remove the station pointed to by Transmit from STATIONS;
    end
until (pointer Transmit points to the first entry on list STATIONS) or (all tasks in list
  TASKS have been covered in the epoch);
```

but the way how stations communicate allows to think of those lists as common for all operational stations.

Two-Lists is structured as a loop (see Algorithm 2). Each iteration of the loop is called an *epoch*. Every *epoch* begins with a transmission schedule and tasks being assigned to processors. During the execution some tasks are performed and if a station transmits the fact of performing them, it is confirmed by removing those certain tasks from list `TASKS`. However, due to adversarial activity some stations may be crashed, but this is recognized as silence heard on the channel in a round that a station was scheduled to transmit. Stations recognized as crashed are also removed from the transmission schedule. Eventually, a new *epoch* begins with updated lists.

Epochs are also structured as loops (see Algorithm 1). Each iteration is now called a *phase*, that consists of three consecutive rounds in which station v:

1. Performs the first unaccomplished task that was assigned to v;

2. v broadcasts one bit, confirming the completion of tasks that were assigned to v, if it was v's turn to broadcast. Otherwise v listens to the channel and attempts to receive a message;

3. Depending on whether a message was heard v updates its information about stations and tasks.

An epoch consists of a number of phases, that is defined by the actual number of operational stations or remaining tasks. In each epoch there is a repeating pattern of phases that consists of the following three rounds: (1) each operational station performs one task. Next (2) a transmission round takes place, where at most one station broadcasts a message, and the rest of the stations attempt to receive it. The process is ended (3) by an updating round, where stations reconstruct their knowledge about operational stations and remaining tasks.

The significance of lists

In the previous section, we mentioned the concept of *knowledge* about stations and tasks, that processors maintain. It was described somehow abstractly, so now we explain it in detail. Furthermore, we provide information on how the stations are scheduled to transmit and how they know, which tasks they should perform.

It is not accidental that the algorithm was named Two-Lists as the most important pieces of information about the system are actually maintained in two lists. The first is list STATIONS. It represents operational (at the beginning of an epoch) processors and sets the order in which stations should transmit in consecutive phases. That list is operated by pointer Transmit, that is incremented after every phase. It points to exactly one station in a single iteration, what prevents collisions on the channel. Hence, when some station did not broadcast we may recognize that it was crashed and should be eliminated from STATIONS, while the pointer is being set to the following processor.

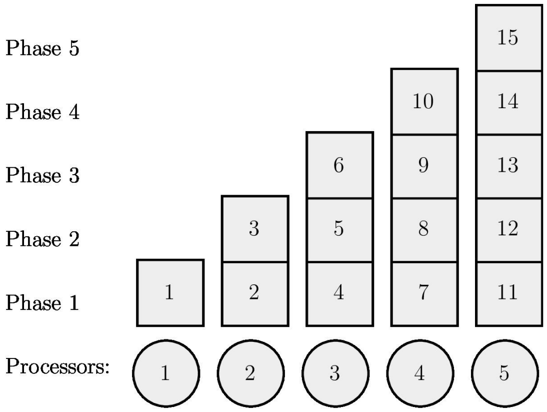

Figure 2.1: Tasks assignment in Two-Lists and a snapshot of an epoch.

The second list is TASKS. It contains remaining tasks, and its associated pointer is Task_To_Do$_v$, separate for each station. Task assignment is organized in the following way (cf., Figure 2.1 for a visualized example). Let us present processors from list STATIONS as a sequence $\langle v_i \rangle_{1 \leq i \leq n}$, where $n = |\texttt{STATIONS}|$ is the number of operational stations at the beginning of the epoch. Each station is responsible for some segment of list TASKS and all segments sum to the whole list. The length of a segment for station v_i equals i in a single epoch. A single task may belong to more than one segment at a time, unless the number of tasks is accordingly greater than the number of stations.

It follows that, if a station broadcasts successfully, then all other stations know, which tasks was it responsible for and may remove them from list TASKS. This allows stations to broadcast just a single bit, in order to prove that they are still operational in an epoch.

It is noticeable that lists STATIONS and TASKS are treated as common to all the devices, because of maintaining common knowledge. However, in fact every station has a private copy of those lists and operates with appropriate pointers.

Finally, there are two additional lists maintained by each station. The first one is list $\texttt{OUTSTANDING}_v$ and it contains the segment of tasks that station v has assigned to perform in an epoch. The second is list $\texttt{DONE}_v$ and it contains tasks already performed by station v. These two additional lists are auxiliary and their main purpose is to structure algorithms in a clear and readable way.

Let us now consider an example epoch in Figure 2.1. The epoch begins with tasks being assigned to each processor. Processors broadcast consecutively in the following phases, since the Two-Lists algorithm is designed as an interlacing sequence of performing tasks and broadcasting progress.

Consequently, processor 1 performs task 1 and at the end of phase 1 performs a broadcast. While processor 1 performs task 1, all other processors also perform their assigned tasks and hence, processor 2 performs task 2, processor 3 performs task 4 and so on. Then phase 2 begins and after performing task 3, processor 2 performs a broadcast, while all other processors also perform their tasks, with an exception of processor 1, which stays idle until the end of the epoch. The epoch ends with the broadcast of the last processor.

Sparse vs dense epochs

The last important element of Two-Lists description, that explains some subtleties are definitions of *dense* and *sparse* epochs.

Definition 1. *Let n = |STATIONS| denote the number of operational stations at the beginning of the epoch. If $n(n+1)/2 \geq$ |TASKS|, then we say that an epoch is dense. Otherwise we say that an epoch is sparse.*

The expression $n(n+1)/2 = 1 + 2 + \cdots + n$ from the definition above determines how many tasks may be performed in a single epoch. If all the broadcasts in Two-Lists are successful, then this is the number of performed (and confirmed) tasks.

In general that is why if we consider a dense epoch, then it is possible that some task i was assigned more than once to different stations. A dense epoch may end when the list of tasks becomes empty. However, for sparse epochs the end condition is consistent with the fact that every station had a possibility to transmit, and pointer `Transmit` passed all the devices on list `STATIONS`.

We end this section with results from [31] stating that Two-Lists is asymptotically work optimal for the channel without collision detection and against the Strongly-Adaptive adversary.

Fact 1. *([31], Theorem 1) Algorithm Two-Lists solves Do-All with* $\mathcal{O}(t+p\sqrt{t}+p\min\{f,t\})$ *work against the f-Bounded adversary, for any* $0 \le f < p$.

Fact 2. *([31], Theorem 2) The f-Bounded adversary, for* $0 \le f < p$, *can force any reliable, possibly randomized, algorithm for the channel without collision detection to perform* $\Omega(t+p\sqrt{t}+p\min\{f,t\})$ *work.*

Fact 3. *([31], Corollary 1) Algorithm Two-Lists is optimal in asymptotic work efficiency, also among randomized reliable algorithms for the channel without collision detection, against the adaptive adversary who may crash all but one station.*

Adversary tactics

Two-Lists complexity formula contains the factor $p\min\{f,t\}$, describing failing work, resulting from the adversary activity. Since the same analysis applies to results presented in this thesis, then we explain it in detail.

If a station crashes in an epoch, then its work in this epoch equals at most the length of the epoch, which is at most p. This is equivalent with the adversary crashing the station just before its intended broadcast. Therefore, failing work is $\mathcal{O}(pf)$, since the adversary can crash f stations at most. Every station performs $\mathcal{O}(t)$ work, because the total number of phases in Two-Lists is at most t. It follows that the failing work is also $\mathcal{O}(pt)$. These two facts together give the estimate $\mathcal{O}(p\min\{f,t\})$ on the magnitude of failing work.

2.3.2 Algorithm Groups-Together

Beside Two-Lists, which serves as a sub-procedure for our considerations, some of our algorithms are built on another algorithm from [31] — Groups-Together. The design of both algorithms is similar, yet Groups-Together was introduced for a channel with collision detection. In what follows, we will describe the technicalities and main differences in this subsection. Let us recall that a shared channel with collision detection provides three types of signals: silence, single and collision, cf., Section 2.1.2.

Simultaneous transmissions are excluded in Two-Lists. In such a case, a simultaneous transmission of multiple stations would result in a silence signal, and does not provide any meaningful information. Because Groups-Together is specifically designed to work on a channel with collision detection, then the feedback from collision signals is extensively used. The main difference is that instead of list `STATIONS`, it maintains list `GROUPS` — and indeed, in Groups-Together stations are arranged into disjoint groups. Assigning stations to groups is as follows. Let n be the smallest number such that $n(n+1)/2 > |\texttt{TASKS}|$ holds. If the overall number of stations on list `GROUPS` at the beginning of an epoch is at least n, then an epoch is *dense*, otherwise it is *sparse*. If we consider any two stations, whose positions on a sorted list of all stations (according to stations' identifiers) are i and j respectively, then they are placed in the same group on list `GROUPS`, if numbers i and j are congruent modulo n. For this reason, any two groups from `GROUPS` differ in size by at most 1. Consequently, the initial partition results in having $\min\{\sqrt{t}, p\}$ balanced groups. Furthermore, such an allocation means that when an epoch is sparse, then groups consist of single stations. Otherwise, some groups contain more than a single station.

Tasks assignment is the same as in Two-Lists, with the difference that now the algorithm operates on groups instead of single stations. In what follows, all stations within a single group have the same tasks assigned and hence, work together on exactly the same tasks. The round-robin schedule of consecutive broadcasts from Two-Lists also applies to Groups-Together, yet now points out particular groups instead of single stations. Consequently, if a group broadcasts simultaneously and there is a collision signal (or a single transmission) heard on the channel, this means that the tasks that the group was responsible for have been actually performed and may be removed from list `TASKS`. However, if silence is heard, then we are sure that all the stations from the group have been crashed.

Apart from the differences described above, Groups-Together is the same as Two-Lists. It is structured as a loop, which one iteration is called an epoch. An epoch is also structured as a repeat loop which single iteration is called a phase. Phases contain three rounds, one of which is for transmission. If no transmission occurs in a phase, we call it *silent*. Otherwise it is called *noisy*. The notions of dense and sparse epochs remain the same as in the Two-Lists analysis.

We finish this section with results from [31] regarding Groups-Together.

Fact 4. *([31], Lemma 4) Algorithm* Groups-Together *is reliable.*

Fact 5. *([31], Theorem 3) Algorithm* Groups-Together *solves Do-All with the minimal work $\mathcal{O}(t + p\sqrt{t})$ against the f-Bounded adversary, for any f such that $0 \leq f < p$.*

2.3.3 The Task-Performing Black Box (TaPeBB)

The algorithms we design in Chapter 4 employ a black-box procedure for arbitrary length tasks that is able to reliably perform a subset of input (in the form of tasks consisting of chains of consecutive subtasks) or report that something went wrong. In what follows, we specify this procedure, called the *Task-Performing Black Box* (TaPeBB for short), and argue that it can be implemented and employed to our considerations.

We use the procedure in both deterministic and randomized solutions. Precisely, all our algorithms in Chapter 4 use TaPeBB, despite the fact that they perform differently in the sense of work complexity.

Most important ideas of our results lie within how to pre-process the input rather than how to actually perform the tasks, so we believe that employing such a black-box improves the clarity of presentation.

General properties of TaPeBB. Let us recall, that in a system with time divided into synchronous slots, that we already called rounds, each round is a possibility for processors to transmit.

The nature of arbitrary length tasks leads, however, to a concept that the time between consecutive broadcasts needs to be adjusted. Specifically, for sets containing long tasks in the non-preemptive configuration of the problem, it may be better to broadcast the fact of performing the task fully, rather than broadcasting partial progress multiple times. Only the final transmission brings valuable information about progress in performing tasks and any intermediate transmissions congest the channel, indicating only that the broadcasting processor is still operational.

Therefore, we assume that TaPeBB has a feature of changing the duration between consecutive broadcasts. We will call the actual time step between consecutive broadcasts a *phase*. Denote the length of a phase by ϕ. Unless stated otherwise, we assume that $\phi = 1$, i.e., the duration of a phase and a round is consistent, thus processors may transmit in any round.

Input: $\texttt{TaPeBB}(v, d, \texttt{TASKS}, \texttt{STATIONS}, \phi)$

- v represents the id of a processor executing TaPeBB.
- TaPeBB takes a list of processors `STATIONS` and a list of tasks `TASKS` as the input, yet from the subtask perspective, i.e., tasks are provided as a chain of subtasks. All

necessary information about subtasks is available through list `TASKS`, including their id's, and how, as well as, in what order they build tasks. It may happen that a task is not done fully and only some initial segment of subtasks forming that task is performed. Because list `TASKS` maintains information about subtasks, such a situation can be successfully handled.

- Additionally, the procedure takes an integer value d. It specifies the number of processors that will be used in the procedure for performing provided task input. The procedure works in such a way that each working processor is responsible for performing a number of tasks. For clarity we assume that TaPeBB always uses the initial d processors from the list of processors. Using a certain number of processors in the procedure allows us to set an upper bound on the amount of work accrued during a single execution.
- Regardless of the value d, if there are processors which did not have any tasks assigned for the execution, then they do not broadcast and so the procedure might end earlier than after d phases.
- We call a single execution an *epoch* and the parameter d is called the length of an epoch (i.e., the number of phases that form an epoch).
- ϕ is the length of a phase i.e., the duration between consecutive broadcasts.

Output: what tasks/subtasks have been done and which processors have crashed.

- Having explained what is the length of an epoch in TaPeBB we now describe the capability of performing subtasks in a single epoch, which is understood as the maximal number of tasks that may be confirmed in an epoch, when it is executed fully without any adversarial distractions. Firstly, let us note, that TaPeBB allows to confirm j subtasks in some round j. This comes from the fact that if a station worked for j rounds and was able to perform one subtask per round, then it can confirm at most j subtasks when it comes to broadcasting in round j. Therefore, the capability of performing subtasks in an epoch is at most $\sum_{j=1}^{d} j$ which is the sum of an arithmetic series with common difference equal 1 over all rounds of an epoch.
- As a result of running a single epoch we have an output information about which subtasks were actually done and whether there were any processors identified as

crashed, when processors were communicating progress (a crash is consistent with a processor being silent when scheduled to broadcast).

In general, the way one can think how TaPeBB works is as follows: processors have a sufficient number of tasks assigned and they interlace performing tasks with communicating progress. Thus, processor 1 performs a single task and transmits the fact of performing it through the channel. In the meantime, all other processors also had the opportunity of performing single tasks. Then, the algorithm proceeds to the following round in which processor 2 performs a task and communicates the fact of performing **two** tasks through the channel. In the following round processor 3 is scheduled to transmit and it is able to perform three tasks by then. In what follows, processors transmit messages one by one in consecutive rounds during a single TaPeBB execution and the procedure terminates. This allows to be sure that a certain number of subtasks/tasks have been performed and identify which processors have crashed.

A candidate algorithm to serve as TaPeBB is a slightly modified Two-Lists algorithm from [31], described in Section 2.3.1. Nevertheless, we assume that it may be substituted with an arbitrary algorithm fulfilling the requirements that we stated above.

Fact 6. TaPeBB *performs at most* $\sum_{j=1}^{d} j$ *subtasks while generating* $p \cdot d \cdot \phi$ *work in each execution.*

Proof. We have at most p processors working for d phases, and the length of each phase is set to ϕ rounds, hence this gives $pd\phi$ units of work in total. □

2.3.4 Consensus tools

In Chapter 6, we use black-box fashioned procedures that allow us to improve the structure of presentation. We now briefly describe their properties and later refer to them in the algorithms' analysis.

Leader-Consensus properties. We use the Leader-Consensus procedure as a black-box tool for reaching Consensus on a small group of processes, and we require that it satisfies the following properties:

- it is executed by a process and takes two values as input: the time for which it is executed (unless it terminates earlier because Consensus was reached) and the current value of a process;

- the output is a tuple (*decided*, *value*), where *decided* is a boolean variable indicating whether the Consensus value has been decided by a process during the procedure and *value* is the current value of a process after the procedure terminates (if the Consensus has been decided – it is the Consensus value);

- it satisfies termination, validity and *conditional agreement*, defined as follows: for any two processes v, w, if Leader-Consensus executed by v outputs (true, x) and Leader-Consensus executed by w outputs (true, y), then $x = y$;

- Leader-Consensus$(T_{LC}(g), x)$ satisfies agreement when run by a group of no more than g processes, with probability at least $\frac{9}{10}$, where T_{LC} is the expected time complexity function of Leader-Consensus.

We say that an algorithm fulfilling properties above satisfies *Conditional-Consensus*. A candidate solution to serve as Leader-Consensus is the Ben-Or and Bar-Joseph's SynRan algorithm from [14], and we refer the reader to the details therein. In particular, to Lemma 4.2 in [14], which proves that SynRan assures conditional agreement besides of other typical properties of Consensus.

Propagate-Msg properties. We assume that procedure Propagate-Msg propagates messages in 1 round with $\mathcal{O}(n^2)$ message complexity. This is consistent with a scenario where full communication takes place and each process sends a message to all the processes.

Chapter 3

Ordered and Delayed adversaries

In this chapter, we consider the Do-All problem in its basic form with unit-length tasks and analyze how different adversarial scenarios influence the difficulty of the problem.

We introduce a hierarchy of adaptive adversaries. The most important parameter of this hierarchy is the partial order, describing adversarial crashes. Other parameters of the hierarchy include the number of crashes f and a delay c in the effect of the adversary's decisions.

We show that adversaries constrained by an order of short width i.e., with short maximal anti-chain or 1-RD adversaries have very little power. This results in the algorithms' performance similar to the one enforced by oblivious adversaries or linearly-bounded adversaries, cf., [31]. More specifically, we develop algorithms ROBAL and GILET, which achieve work performance close to the absolute lower bound $\Omega(t + p\sqrt{t})$ against "narrow-ordered" and 1-RD adversaries, respectively.

In case of ordered adversaries restricted by orders of arbitrary width $k \leq f$, we present algorithm GRUTECH that guarantees work $\mathcal{O}\left(t + p\sqrt{t} + p\min\left\{\frac{p}{p-f}, t, k\right\}\log p\right)$ against Ordered adversaries restricted by orders of width k, and show that it is efficient by proving a lower bound for a broad class of partial orders. Table 3 presents detailed results and comparisons.

The hierarchy of Ordered and Round-Delayed adversaries is illustrated in Figure 3.1. It depends on three main factors. Additionally, we introduce several solutions for the specified settings. Consequently, our contribution is a complement to adversarial scenarios presented in the literature together with a taxonomy describing the dependencies between different adversaries.

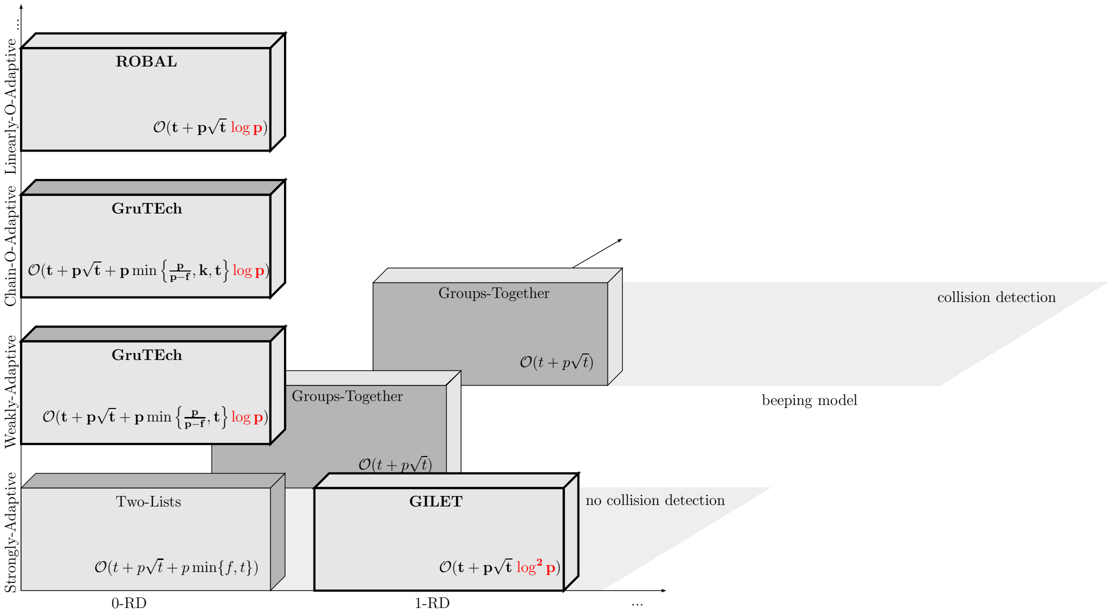
Linearly-O-Adaptive
...
ROBAL
$\mathcal{O}(\mathbf{t}+\mathbf{p}\sqrt{\mathbf{t}}\log\mathbf{p})$
Chain-O-Adaptive
GruTEch
$\mathcal{O}(\mathbf{t}+\mathbf{p}\sqrt{\mathbf{t}}+\mathbf{p}\min\left\{\frac{\mathbf{p}}{\mathbf{p-f}},\mathbf{k},\mathbf{t}\right\}\log\mathbf{p})$
Weakly-Adaptive
GruTEch
$\mathcal{O}(\mathbf{t}+\mathbf{p}\sqrt{\mathbf{t}}+\mathbf{p}\min\left\{\frac{\mathbf{p}}{\mathbf{p-f}},\mathbf{t}\right\}\log\mathbf{p})$
Strongly-Adaptive
Two-Lists
$\mathcal{O}(t+p\sqrt{t}+p\min\{f,t\})$
Groups-Together
$\mathcal{O}(t+p\sqrt{t})$
Groups-Together
$\mathcal{O}(t+p\sqrt{t})$
collision detection
beeping model
GILET
$\mathcal{O}(\mathbf{t}+\mathbf{p}\sqrt{\mathbf{t}}\log^{\mathbf{2}}\mathbf{p})$
no collision detection
0-RD
1-RD
...

Figure 3.1: The hierarchy of adversaries for the Do-All problem.

First of all, we have the vertical axis which describes adversary features, that is how restricted its decisions are. We have the Strongly-Adaptive adversary in the origin, who may decide online which stations will be crashed. Above the Strongly-Adaptive adversary is the Weakly-Adaptive adversary, who is slightly weaker and has to declare the subset of stations that will be prone to crashes before the algorithm execution. Next, we have the k-Chain-Ordered adversary and its more general version k-Ordered adversary (Chain-O-Adaptive in Figure 3.1 for consistency). Apart from declaring the faulty subset, the former adversary is restricted by partial orders being collections of disjoint k chains, while the latter — by all partial orders of thickness k (and so k-chains as well). Finally, there is the Linearly-Ordered adversary (Linearly-O-Adaptive in Figure 3.1) — its order is described by a linear pattern of processor crashes.

The horizontal axis describes another feature of the adversary, that we consider in this chapter i.e., the Round-Delay of adversary decisions. Similarly, the configuration for the problem is hardest in the origin and a 0-RD adversary is the strongest against which we may execute an algorithm. An interesting particularity is that if the Strongly-Adaptive adversary's decisions are delayed by at least one round, then we may design a solution, the

algorithm	channel	adv.	work	ref.	lower bound	ref.
Two-Lists	no-CD	SA	$\mathcal{O}(t + p\sqrt{t} + p\min\{f, t\})$	[31] Thm 1	$\Omega(t + p\sqrt{t} + p\min\{f, t\})$	[31] Thm 2
Groups-Together	CD	SA	$\mathcal{O}(t + p\sqrt{t})$	[31] Thm 3	$\Omega(t + p\sqrt{t})$	[31] Lem 2
Mix-Rand	no-CD	WALB	$\mathcal{O}(t + p\sqrt{t})$	[31] Thm 5	$\Omega(t + p\sqrt{t})$	[31] Lem 2
ROBAL	no-CD	LOA	$\mathcal{O}(t + p\sqrt{t}\log p)$	Sec. 3.1	$\Omega(t + p\sqrt{t})$	[31] Lem 2
GruTEch	no-CD	WA	$\mathcal{O}(t + p\sqrt{t} + p\min\left\{\frac{p}{p-f}, t\right\}\log p)$	Sec. 3.2	$\Omega(t + p\sqrt{t} + p\min\left\{\frac{p}{p-f}, t\right\})$	[31] Thm 6
GruTEch	no-CD	COA	$\mathcal{O}(t + p\sqrt{t} + p\min\left\{\frac{p}{p-f}, t, k\right\}\log p)$	Sec. 3.3	$\Omega(t + p\sqrt{t} + p\min\left\{\frac{p}{p-f}, k, t\right\})$	Sec. 3.3
GILET	no-CD	1-RD	$\mathcal{O}(t + p\sqrt{t}\log^2 p)$	Sec. 3.4	$\Omega(t + p\sqrt{t})$	[31] Lem 2

Table 3.1: Summary of main results in Chapter 3; first three were introduced in [31], the other are presented in this chapter. CD stands for collision detection model feature. SA stands for Strongly-Adaptive adversary, WA (WALB) stands for Weakly-Adaptive (Linearly-Bounded) adversary, COA stands for Chain-Ordered adversary, LOA stands for Linearly-Ordered adversary, and 1-RD stands for 1-Round-Delay adversary. Underlined algorithms are randomized and their corredponding complexity formulae hold in expectation.

work complexity of which is independent of the number crashes.

The axis orthogonal to those already considered, describes the channel feedback. In the origin we have a multiple-access channel without collision detection, then there is the beeping channel (cf., Section 2.1.2), followed by MAC with collision detection.

We may see that the most difficult setting is in the origin, while the further from the origin, the easier the problem becomes. Boxes in Figure 3.1 represent algorithms and their work complexities in certain configurations of the model i.e. features described above. Bold boxes denote algorithms from this chapter and the remaining ones are from [31]. Factors marked red denote the "distance" from the lower bounds, understood as how far the algorithms are from optimum.

A subset of the results from this chapter forms a hierarchy of partially ordered adversaries. The first solution we introduced was designed to work against a Linearly-Ordered adversary, whose pattern of crashes is described by a linear order. The upper bound of this algorithm does not depend on the number of crashes and is just logarithmically far from the minimal work complexity in the model under consideration. The second algorithm serves for the case when the adversary's partial order of stations forms a maximum length anti-chain. Nevertheless, we also analyze this solution against an in-between situation when the partial order consists of k chains of arbitrary lengths, yet the sum of their lengths is f.

Due to the specific nature of the Do-All problem on a multiple-access channel, the work complexity formulae presented in this chapter consist of a sum of several factors, which represent different cases of the analysis. A reliable randomized algorithm is required to have the property that all the tasks are eventually performed in all executions, hence randomization may contribute to efficiency but must not violate correctness. Hence, even if some cases of the algorithms' analyses hold with high probability, we still need to handle complementary events, which happen with some small probability. Therefore, it is more appropriate to analyze results from this chapter in expectation.

In order to conclude this chapter, we would like to emphasize that building on solutions from [31] we introduce different algorithms and specific adversarial scenarios for more complex setups. This, to some extent, filled the gap from [31] for randomized algorithms solving Do-All in the most challenging adversarial scenarios and communication channels providing least feedback.

3.1 ROBAL — Random Order Balanced Allocation Lists

In this section we describe and analyze the algorithm for the Do-All problem in presence of a Linearly-Ordered adversary on a channel without collision-detection. Its expected work complexity is $\mathcal{O}(t + p\sqrt{t}\log p)$ and it uses the Two-Lists procedure from [31] (cf., Section 2.3.1).

ROBAL (Algorithm 3) from perspective of some station v, works in such a way that, initially, v checks whether $p \leq \sqrt{t}$, because for such parameters it can execute the Two-Lists algorithm – its complexity is linear in t (see Fact 8 in Section 3.1.1) for such paramters. If this is not the case, the main body of the algorithm is executed, yet another specific condition is checked: $\log p > e^{\frac{\sqrt{t}}{32}}$, which is for handling a scenario with a very small number of tasks and is specifically required for the algorithm analysis. If so, all tasks are assigned to each station. If every station has all the tasks assigned, then after t phases we may be sure that all the tasks are done, because always at least one station remains operational. Because of the specific range of the parameters, redundant work in this case is acceptable (i.e., within the claimed bound) from the point of view of our analysis. However, we execute procedure Confirm-Work (Algorithm 4) in order to confirm this fact on the channel.

Confirm-Work is a type of leader election procedure. It assigns a certain probability of a station to broadcast, so we expect that exactly one station will transmit in a number of trials. Because we cannot be sure what is the actual number of operational stations, the probability is changed multiple times until all tasks are confirmed.

If the specific conditions discussed above are not satisfied (Algorithm 3 lines 1-10), then Mix-And-Test is executed (Algorithm 5). It changes the order of stations on list `STATIONS`.

Algorithm 3: ROBAL, pseudo-code for processor v

```
if $p \leq \sqrt{t}$ // $p$ and $t$ are global parameters known by the system
then
    execute Two-Lists;
end
else
    initialize STATIONS to a sorted list of all $p$ names of stations;
    if $\log p > e^{\frac{\sqrt{t}}{32}}$ then
        execute a $t$-phase epoch with all tasks assigned (without transmissions);
        execute Confirm-Work;
    end
    else
        $i = 0$;
        repeat
            if $(\frac{p}{2^i} \leq \sqrt{t})$ then
                Execute Two-Lists;
            end
            if Mix-And-Test$(i, t, p)$ then
                repeat
                    Execute $\sqrt{t}$ phases of Two-Lists;
                until less than $\frac{1}{4}\sqrt{t}$ broadcasts are heard;
            end
            increment $i$ by 1;
        until $i = \lceil \log p \rceil$;
    end
end
```

Algorithm 4: Confirm-Work, pseudo-code for processor v

```
$i := 0$ ;
repeat
    coin := $\frac{p}{2^i}$ ;
    toss a coin with the probability $\texttt{coin}^{-1}$ of heads to come up;
    if heads came up in the previous step then
        broadcast $v$ via the channel and attempt to receive a message;
    end
    if some station $w$ was heard then
        clear list TASKS;
        break;
    end
    else
        increment $i$ by 1 ;
        if  $i = \lceil \log p \rceil + 1$ then
            $i := 0$;
        end
    end
until a broadcast was heard;
```

Precisely, stations that performed successful broadcasts are moved to the front of that list. This procedure has two purposes. First, changing the order makes the adversary less flexible in crashing stations, as its order is already determined. Second, we may predict with high probability to which interval $n \in (\frac{p}{2^i}, \frac{p}{2^{i-1}}]$ for $i = 1, \cdots, \lceil \log p \rceil$ the current number of operational stations belongs, which is important from the work analysis perspective.

Stations moved to the front of list STATIONS are called *leaders.* All operational stations execute the same code so, by symmetry, we expect that elected *leaders* will be uniformly distributed in the adversary's order between stations that were not chosen as *leaders.* This allows us to assume that a crash of a *leader* is likely to be preceded by several crashes of other stations (see Figure 3.2).

Let us consider procedure Mix-And-Test in detail. If n is the previously predicted number of operational stations, then each of the stations tosses a coin with the probability of success equal to $1/n$. In case where none or more than one of the stations broadcasts then silence is heard on the channel, as there is no collision detection. Otherwise, when only one station did successfully broadcast it is moved to the front of list STATIONS and the procedure starts again with a decremented parameter. However, stations that have already

Algorithm 5: Mix-And-Test, pseudo-code for processor v

```
Input: $i, t, p$
coin $:= \frac{p}{2^i}$;
for $\sqrt{t} \log p$ times do
    if $v$ has not been moved to the front of list STATIONS yet then
        toss a coin with the probability coin$^{-1}$ of heads to come up
    end
    if heads came up in the previous step then
        broadcast $v$ via the channel and attempt to receive a message
    end
    if some station $w$ was heard then
        move station $w$ to the front of list STATIONS;
        decrement coin by 1;
    end
end
if at least $\sqrt{t}$ broadcasts were heard during the loop then
    return true;
end
else
    return false;
end
```

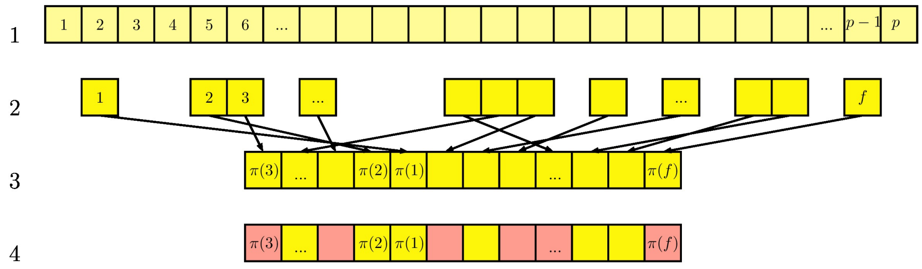
1
2
3
4
1 2 3 4 5 6 ... ... $p-1$ p
1 2 3 ... ... f
$\pi(3)$ ... $\pi(2)$ $\pi(1)$ ... $\pi(f)$
$\pi(3)$ ... $\pi(2)$ $\pi(1)$ ... $\pi(f)$

Figure 3.2: (1) Initially we have p stations. (2) The adversary chooses f stations prone to crashes. (3) Then it declares the order according to which stations will crash. (4) Mix-And-Test chooses a number of leaders which are expected to be distributed uniformly among the adversary linear order.

been moved to the front do not take part in the following iterations of the procedure.

Upon having chosen the leaders, regular work is performed. An important feature of our algorithm is that we do not perform full epochs, but only $\sqrt{t}$ phases of each Two-Lists epoch. This allows us to be sure that the total work accrued in each epoch does not exceed $p\sqrt{t}$. If, at some point, the number of successful broadcasts substantially drops, another Mix-And-Test (Algorithm 5) procedure is executed and a new set of leaders is chosen.

Before the algorithm execution the Linearly-Ordered adversary has to choose f stations prone to crashes and declare a sequence, that will describe in what order those crashes may happen. In what follows, when there are unsuccessful broadcasts of *leaders* (crashes) we may be approaching the case when $n \leq \sqrt{t}$ and we can execute Two-Lists whose complexity is linear in t for such parameters. Alternatively, the adversary spends the majority of its possible crashes and stations may finish all the tasks without any distractions.

3.1.1 Analysis of ROBAL

We begin our analysis with a general statement about the reliability of ROBAL.

Lemma 2. *Algorithm* ROBAL *is reliable.*

Proof. We need to show that all the tasks will be performed as a result of executing the algorithm. First of all, if the algorithm falls into the case where $\frac{p}{2^i} \leq \sqrt{t}$ (or initially $p \leq \sqrt{t}$), then Two-Lists is executed, which is reliable, as we know from [31].

Secondly, when $\log p > e^{\frac{\sqrt{t}}{32}}$, then all tasks are assigned to every station and they all work for t phases. We know that $f < p$ so at least one station will perform all the tasks.

Finally, if those conditions do not hold, the algorithm runs an external loop (Algorithm 3, line 13) in which variable i increments after each iteration. If the loop is performed $\lceil \log p \rceil$ times, then we run Two-Lists. Variable i may not be incremented only if the algorithm enters and stays in the internal loop (Algorithm 3, line 18). However this is possible only after performing all the tasks, because the internal loop runs for a constant number of times until all tasks are completed. □

We now proceed to a statement bounding the worst case work of Two-Lists, which is used as a sub-procedure in ROBAL.

Fact 7. Two-Lists *always solves the Do-All problem with* $\mathcal{O}(pt)$ *work.*

Proof. Let us recall that the amount of work resulting from adversarial crashes in Two-Lists is $p \min\{f, t\}$. Hence, there are at most $\min\{f, t\}$ crashes, which can happen in at most $\min\{f, t\}$ distinct phases. On the other hand, Two-Lists is always executed against an adversary who can crash all but one station. This means that in each epoch there will be at least one phase with a successfull transmission, confirming the completion of at least one task. Thus, the total number of phases of Two-Lists is at most $t + \min\{f, t\} = \mathcal{O}(t)$. Since there are at most p operational stations during those $\mathcal{O}(t)$ phases, we obtain $\mathcal{O}(pt)$ work. □

We already mentioned that ROBAL was modeled in such a way that, whenever $\frac{p}{2^i} \leq \sqrt{t}$ holds, the Two-Lists algorithm is executed, because the work complexity of Two-Lists for such parameters is $\mathcal{O}(t)$. We prove it in the following fact.

Fact 8. *Let n be the number of operational processors, and t be the number of remaining tasks. Then for $n \leq \sqrt{t}$* Two-Lists *work complexity is $\mathcal{O}(t)$.*

Proof. If $n \leq \sqrt{t}$, then the maximal number of possible crashes is $f < n$, hence $f < \sqrt{t}$. Algorithm Two-Lists has $\mathcal{O}(t + p\sqrt{t} + p\ \min\{f, t\})$ work complexity. In follows that the complexity is $\mathcal{O}(t + \sqrt{t}\sqrt{t} + \sqrt{t}\ \min\{\sqrt{t}, t\}) = \mathcal{O}(t)$. □

Figure 3.2 presents the way how we expect leaders to interlace other stations in the adversary's order. The following lemma estimates the probability that if a number of leaders was crashed, then, overall, a significant number of stations must have been crashed as well.

Lemma 3. *Let us assume that we have n operational stations at the beginning of an epoch, $\sqrt{t}$ of which were chosen leaders. If the adversary crashes $n/2$ stations, then the probability that there were $3/4$ of the overall number of leaders crashed in this group does not exceed $e^{-\frac{1}{8}\sqrt{t}}$.*

Proof. We have n stations, among which $\sqrt{t}$ are leaders. The adversary crashes $n/2$ stations and our question is how many leaders were in this group?

The hypergeometric distribution function with parameters N - number of elements, K - number of highlighted elements, l - number of trials, k - number of successes, is given by:

$$\mathbb{P}[X = k] = \frac{\binom{K}{k}\binom{N-K}{l-k}}{\binom{N}{l}}.$$

The following tail bound from [56] tells us, that for any $s > 0$ and $p = \frac{K}{N}$:

$$\mathbb{P}[X \geq (p+s)l] \leq e^{-2s^2 l}.$$

Identifying this with our process we have that $K = n/2$, $N = n$, $l = \sqrt{t}$ and consequently $p = 1/2$. Placing $s = 1/4$ we have that

$$\mathbb{P}\left[X \geq \frac{3}{4}\sqrt{t}\right] \leq e^{-\frac{1}{8}\sqrt{t}}.$$

□

The following two lemmas give us the probability that Mix-And-Test diagnoses the number of operational stations correctly with high probability. This gives us a bound for the expected work of ROBAL in Theorem 1. However, let us emphasize, that if Mix-And-Test is unsuccessful, this does not violate the reliability of the whole algorithm, yet influences the work complexity of the pessimistic case.

Lemma 4. *Let us assume that the number of operational stations is in the $(\frac{p}{2^i}, \frac{p}{2^{i-1}}]$ interval. Then procedure* Mix-And-Test*(i, t, p) returns true with probability $1 - e^{-c\sqrt{t}\log p}$, for some $0 < c < 1$.*

Proof. We begin with proving the following claim.

Claim 1. *Let the current number of operational stations be in interval $(\frac{x}{2}, x]$. Then the probability of the event that in a single iteration of* Mix-And-Test *exactly one station will broadcast is at least $\frac{1}{2\sqrt{e}}$ (where the $\texttt{coin}^{-1}$ parameter is $\frac{1}{x}$).*

Proof of the Claim. Let us consider a scenario where the number of operational stations is in interval $(\frac{x}{2}, x]$ for some x. If every station broadcasts with probability of success equal to $1/x$ then the probability of an event that exactly one station transmits is $(1-\frac{1}{x})^{x-1} \geq 1/e$. Estimating the worst case, when there are $\frac{x}{2}$ operational stations (and the probability of success remains $1/x$) we have that

$$\frac{1}{2}\left(1-\frac{1}{x}\right)^{x\cdot\frac{x-2}{2}} \geq \frac{1}{2\sqrt{e}}\ .$$

This concludes the proof of the Claim. □

According to the Claim, the probability of an event that in a single round of Mix-And-Test exactly one station will be heard is $\frac{1}{2\sqrt{e}}$.

We assume that $n \in (\frac{p}{2^i}, \frac{p}{2^{i-1}}]$. Let us show that the algorithm confirms appropriate i with probability $1-e^{-c\sqrt{t}\log p}$. For this purpose, we need $\sqrt{t}$ transmissions to be heard.

Let X be a random variable such that $X = X_1 + \cdots + X_{\sqrt{t}\log p}$, where $X_1, \cdots, X_{\sqrt{t}\log p}$ are Poisson trials and

$$X_k = \begin{cases} 1 & \text{if station broadcasted,} \\ 0 & \text{otherwise.} \end{cases}$$

We know that

$$\mu = \mathbb{E}X = \mathbb{E}X_1 + \cdots + \mathbb{E}X_{\sqrt{t}\log p} \geq \frac{\sqrt{t}\log p}{2\sqrt{e}}.$$

To estimate the probability that $\sqrt{t}$ transmissions were heard we use the Chernoff's inequality.

We want to have that $(1-\epsilon)\mu = \sqrt{t}$. Thus $\epsilon = \frac{\mu-\sqrt{t}}{\mu} = \frac{\log p - 2\sqrt{e}}{\log p}$ and $0 < \epsilon < 1$ for sufficiently large p. Hence

$$\mathbb{P}[X < \sqrt{t}] \quad \leq \quad e^{-\frac{\left(\frac{\log p - 2\sqrt{e}}{\log p}\right)^2}{2}\frac{\sqrt{t}\log p}{2\sqrt{e}}} = e^{-c\sqrt{t}\log p}\ ,$$

for some bounded $0 < c < 1$. We conclude that with probability $1-e^{-c\sqrt{t}\log p}$ we confirm the correct i which describes and estimates the current number of operational stations. □

Lemma 5. *If there are more than* $\frac{p}{2^{i-1}}$ *operational stations, then* Mix-And-Test(i, t, p)

returns false with probability not less than $1-(\log p)^2\ \max\{e^{-\frac{1}{8}\sqrt{t}}, e^{-c\,\sqrt{t}\log p}\}$.

Proof. Let A_i denote an event that at the beginning of an execution of the Mix-And-Test(i, t, p) procedure there are no more than $\frac{p}{2^{i-1}}$ operational stations.

The basic case when $i=0$ is trivial, because initially we have p operational stations, thus $\mathbb{P}(A_0)=1$. Let us consider an arbitrary i. We know that

$$\mathbb{P}(A_i)=\mathbb{P}(A_i|A_{i-1})\mathbb{P}(A_{i-1})+\mathbb{P}(A_i|A_{i-1}^c)\mathbb{P}(A_{i-1}^c)\geq\mathbb{P}(A_i|A_{i-1})\mathbb{P}(A_{i-1}).$$

Let us estimate $\mathbb{P}(A_i|A_{i-1})$. Conditioned on that event A_{i-1} holds, we know that after executing Mix-And-Test$(i-1,t,p)$ we had no more than $\frac{p}{2^{i-2}}$ operational stations. Hence, if we are now considering Mix-And-Test(i,t,p), then we have two options:

1. Mix-And-Test$(i-1,t,p)$ returned *false*,

2. Mix-And-Test$(i-1,t,p)$ returned *true*.

Let us examine what do these cases mean:

1. If the procedure returned *false* then we know from Lemma 4 that with probability $1-e^{-c\,\sqrt{t}\log p}$ there had to be no more than $\frac{p}{2^{i-1}}$ operational stations. If that number would be in the interval $(\frac{p}{2^{i-1}},\frac{p}{2^{i-2}}]$ then the probability of returning *false* would be less than $e^{-c\,\sqrt{t}\log p}$.

2. If the procedure returned *true*, this means that when executing it with parameters $(i-1,f,p)$ we had no more than $\frac{p}{2^{i-1}}$ operational stations. Then the internal loop of ROBAL was broken, so according to Lemma 3 we conclude that the overall number of operational stations had to reduce by half with probability at least $1-e^{-\frac{1}{8}\sqrt{t}}$.

Consequently, we deduce that $\mathbb{P}(A_i|A_{i-1})\geq(1-\max\{e^{-\frac{1}{8}\sqrt{t}},e^{-c\,\sqrt{t}\log p}\})$. Hence $\mathbb{P}(A_i)\geq(1-\max\{e^{-\frac{1}{8}\sqrt{t}},e^{-c\,\sqrt{t}\log p}\})^i$. Together with the fact, that $i\leq\log p$ and the Bernoulli inequality we have that

$$\mathbb{P}(A_i)\geq1-\log p\ \max\{e^{-\frac{1}{8}\sqrt{t}},e^{-c\,\sqrt{t}\log p}\}.$$

We conclude that the probability that the conjunction of events $A_1,\cdots,A_{\log p}$ holds is at least

$$\mathbb{P}\left(\bigcap_{i=1}^{\log p} A_i\right) \geq 1 - (\log p)^2 \ \max\{e^{-\frac{1}{8}\sqrt{t}}, e^{-c\,\sqrt{t}\log p}\},$$

what ends the proof.

□

We can now proceed to the main result of this section.

Theorem 1. ROBAL *performs $\mathcal{O}(t + p\sqrt{t}\log p)$ expected work against the Linearly-Ordered adversary on the channel without collision detection.*

Proof. In the algorithm we are constantly controlling whether condition $\frac{p}{2^i} > \sqrt{t}$ holds. If not, then we execute Two-Lists whose complexity is $\mathcal{O}(t)$ for such parameters.

If this condition does not hold initially then we check another one i.e., whether $\log p > e^{\frac{\sqrt{t}}{32}}$ holds. For such configuration we assign all the tasks to every station. Work accrued during such a procedure is $\mathcal{O}(pt)$. However when $\log p > e^{\frac{\sqrt{t}}{32}}$ then together with the fact that $e^x > x^2/2$ we have that $t = \mathcal{O}(\log p)$ and consequently the total complexity is $\mathcal{O}(p \log p)$.

Finally, successful stations, that performed all the task have to confirm this fact. We demand that only one station will transmit and if this happens, the algorithm terminates. The expected value of a geometric random variable lets us assume that this confirmation will happen in an expected number of $\mathcal{O}(\log p)$ rounds, generating $\mathcal{O}(p \log p)$ work.

When none of the conditions mentioned above hold, we proceed to the main part of the algorithm. The testing procedure by Mix-And-Test for each of disjoint cases, where $n \in (\frac{p}{2^i}, \frac{p}{2^{i-1}}]$ requires a certain amount of work that can be estimated by $\mathcal{O}(p\sqrt{t}\log p)$, as there are $\sqrt{t}\log p$ testing phases in each case and at most $\frac{p}{2^i}$ stations take part in a single testing phase for a certain case.

In the algorithm, we run through disjoint cases where $n \in (\frac{p}{2^i}, \frac{p}{2^{i-1}}]$. From Lemma 3 we know that when some of the leaders were crashed, then a proportional number of all the stations had to be crashed. When leaders are crashed but the number of operational stations still remains in the same interval, then the lowest number of tasks will be confirmed if only the initial segment of stations will transmit. As a result, when half of the leaders were crashed, then the system still confirms $\frac{t}{8} = \Omega(t)$ tasks. This means that even if so many crashes occurred, $\mathcal{O}(1)$ epochs still suffice to do all the tasks. Summing work over all the cases may be estimated as $\mathcal{O}(p\sqrt{t})$.

By Lemma 5 we conclude that the expected work complexity is bounded by:

$$\Big((\log p)^2 \ \max\{e^{-\frac{1}{8}\sqrt{t}}, e^{-c\,\sqrt{t}\log p}\}\Big)\,\mathcal{O}(pt + p\sqrt{t}\log^2 p)$$

$$+\Big(1-(\log p)^2 \ \max\{e^{-\frac{1}{8}\sqrt{t}}, e^{-c\,\sqrt{t}\log p}\}\Big)\,\mathcal{O}(p\sqrt{t}\log p) = \mathcal{O}(p\sqrt{t}\log p),$$

where the first expression comes from the fact, that if we entered the main loop of the algorithm then we know that we are in a configuration where $\log p \le e^{\frac{\sqrt{t}}{32}}$. Thus we have that

$$\frac{pt + p\sqrt{t}\log^2 p}{e^{\frac{\sqrt{t}}{8}}} \le \frac{pt + pt\log^2 p}{e^{\frac{\sqrt{t}}{16}}e^{\frac{\sqrt{t}}{16}}} \le \frac{p + p\log^2 p}{e^{\frac{\sqrt{t}}{16}}} \le p + p\log p = \mathcal{O}(p\log p),$$

what ends the proof. □

3.2 GruTEch — Groups Together with Echo

In this section, we present a randomized algorithm designed to reliably perform Do-All in presence of a Weakly-Adaptive adversary on a shared channel without collision detection. Its expected work complexity is $\mathcal{O}(t + p\sqrt{t} + p\ \min\{p/(p-f), t\}\log p)$.

3.2.1 Description of GruTEch

Our solution is built on algorithm Groups-Together (details in Section 2.3.2) and a Crash-Echo procedure that works as a kind of fault-tolerant replacement of the collision detection mechanism (which is not present in the model). In fact, the algorithm presented here is asymptotically only logarithmically far from matching the lower bound shown in [31], which, to some extent, answers the open question stated therein.

The Crash-Echo procedure. Let us recall the details of Groups-Together from Section 2.3.2. All the stations within a certain group have the same tasks assigned and when it comes to transmitting they do it simultaneously. This strongly relies on the collision detection mechanism, as the stations do not necessarily need to know which station transmitted, but they need to know that there is progress in performing tasks. That is why if a collision is heard and all the stations within the same group were doing the same tasks, we can deduce that those tasks were actually done.

Algorithm 6: GruTEch; pseudo-code for processor v

```
initialize STATIONS to a sorted list of all $p$ stations;
arrange all $p$ names of stations into list GROUPS of groups;
initialize both TASKS and $\texttt{OUTSTANDING}_v$ to sorted list of all $t$ names of tasks;
initialize $\texttt{DONE}_v$ to an empty list of tasks;
initialize $i := 0$ ;
initialize $leader$ := Elect-Leader($i$) and add the $leader$ to each group;
repeat
    Epoch-Groups-CE($i$);
until $halted$;
```

In our model we do not have collision detection, however we designed a mechanism that provides the same feedback without contributing too much work to the algorithm's complexity. Strictly speaking we begin with choosing a leader. Its work will be of a dual significance. On one hand it will belong to some group and perform tasks regularly. But on the other hand it will also perform additional transmissions in order to indicate whether there was progress when stations transmitted.

When a group of stations is indicated to broadcast the Crash-Echo procedure is executed. It consists of two rounds where the whole group transmits together with the leader in the first one and in the second only the leader transmits. We may hear two types of signals:

- **single** - a legible, single transmission was heard. Exactly one station transmitted.
- **silence** - a signal indistinguishable from the background noise is heard. None or more than one station transmitted.

Let us examine what are the possible pairs (group & leader, leader) of signals heard in such approach:

- **(silence, single)** - in the latter round the leader is operational, so it must have been operational in the former round. Because silence was heard in the former round this means that there was a successful transmission of at least two stations one of which was the leader. This is a fully successful case.
- **(single, single)** - the former and the latter round were single, so we conclude that it was the leader who transmitted in both rounds. If the leader belonged to the group

Algorithm 7: Epoch-Groups-CE; pseudo-code for processor v

```
set pointer Task_To_Do$_v$ on list TASKS to the initial position of the range $v$;
set pointer Transmit to the first item on list GROUPS;
repeat
    // Round 1:
    perform the first task on list TASKS, starting from the one pointed to by
      Task_To_Do$_v$, that is in list OUTSTANDING$_v$;
    move the performed task from list OUTSTANDING$_v$ to list DONE$_v$;
    advance pointer Task_To_Do$_v$ by one position on list TASKS;
    // Rounds 2 & 3:
    if Transmit points to $v$ then
        execute Crash-Echo;
    end
    attempt to receive a pair of messages;
    // Round 4:
    if (silence, single) was heard in the preceding round then
        let $w$ be the first station in the group pointed to by Transmit:;
        for each item $x$ on list DONE$_w$ do
            if $x$ is on list OUTSTANDING$_v$ then
                move $x$ from OUTSTANDING$_v$ to DONE$_v$;
            end
            if $x$ is on list TASKS then
                remove $x$ from TASKS;
            end
        end
        if list TASKS is empty then
            halt;
        end
        advance pointer Transmit by one position on list GROUPS;
    end
    else
        if (single, single) was heard in the preceding round then
            remove the group pointed to by Transmit from GROUPS;
        end
        else
            remove leader from all the groups on list GROUPS;
            leader := Elect-Leader($i$) and add the leader to each group;
        end
    end
until pointer Transmit points to the first entry on list GROUPS;
rearrange all stations in the groups of list GROUPS into a new version of list GROUPS;
```

Algorithm 8: Crash-Echo; pseudo-code for processor v

```
// Round 1:
broadcast one bit;
// Round 2:
if $v = leader$ then
    broadcast one bit;
end
```

Algorithm 9: Elect-Leader; pseudo-code for processor v

```
Input: i
repeat
    coin := $p$;
    toss a coin with the probability coin$^{-1}$ of heads to come up;
    if heads came up in the previous step then
        broadcast $v$ via the channel and attempt to receive a message;
    end
    if some station $w$ was heard then
        $leader := w$;
        return $leader$;
    end
    else
        increment $i$ by 1;
    end
until $i < p$;
set pointer Transmit$_{STATIONS}$ to the first item on list STATIONS;
repeat
    if Transmit$_{STATIONS}$ points to $v$ then
        broadcast one bit;
        attempt to receive a message;
        if some station $w$ was heard then
            $leader := w$;
            return $leader$;
        end
    end
    advance pointer Transmit$_{STATIONS}$ by one position on list STATIONS;
until a transmission was heard;
```

scheduled to transmit, then we have progress; otherwise not.

- **(silence, silence)** - if both rounds were silent we cannot be sure whether there was any progress. Additionally, we need to elect a new leader.
- **(single, silence)** - when the former round was single we cannot be sure whether the tasks were performed; a new leader needs to be chosen.

The Weakly-Adaptive adversary has to declare some f stations that are prone to crashes. The elected leader might belong to that subset and be crashed at some time. When this is examined, the algorithm has to switch to the Elect-Leader mode, in order to select another leader. Consequently, the most significant question from the point of view of the algorithm's analysis is what is the expected number of trials to choose a non-faulty leader.

Two modes. We need to select a leader and be sure that it is operational in order to have our progress indicator working instead of the collision detection mechanism. When the leader is operational we simply run Groups-Together algorithm with the difference that instead of a simultaneous transmission by all the stations within a group, we run the Crash-Echo procedure that allows us to distinguish whether there was progress.

Choosing the leader is performed by procedure Elect-Leader, where each station tosses a coin with the probability of success equal $1/p$. If a station is successful then it transmits in the following round. If exactly one station transmits, then the leader is chosen. Otherwise, the experiment is continued (for p rounds in total). Nevertheless, if this still does not work, then the first station that transmits in a round-robin fashion procedure, becomes the leader.

Note that we have a special variable i used as a counter that is incremented in the Elect-Leader procedure until it reaches value p. We assume that this value is passed to Elect-Leader by reference, so that its incrementation is also recognized in the main body of the Epoch-Groups-CE algorithm, thus i is a global counter.

3.2.2 Analysis of GruTEch

Let us begin the analysis of GruTEch by recalling an important result from [31].

Theorem 2. *([31], Theorem 6) The Weakly-Adaptive f-Bounded adversary can force any reliable randomized algorithm solving Do-All in the channel without collision detection to perform* $\Omega(t + p\sqrt{t} + p\ \min\{p/(p-f), t\})$ *expected work.*

In fact the theorem above in [31] stated that the lower bound was $\Omega(t+p\sqrt{t}+p\ \min\{f/(p-f),t\})$, however the proof relied on the expected number of rounds before a successful transmission took place and the authors did not take into consideration that the first successful transmission may occur earliest in round 1. Hence as it must be at least round number 1, we correct it as follows: $\frac{f}{p-f}+1=\frac{f}{p-f}+\frac{p-f}{p-f}=\frac{p}{p-f}$.

Lemma 6. GruTEch *is reliable.*

Proof. The reliability of GruTEch is a consequence of the reliability of Groups-Together. We do not make any changes in the core of the algorithm. Crash-Echo does not affect the algorithm, as it always finishes. Elect-Leader procedure always finishes as well. The first loop is executed for at most p times and then it ends. The second loop awaits to hear a broadcast in a round-robin manner. But we know that $0 \leq f \leq p-1$, so always one processor remains operational and it will respond. □

Let us define a *sustainable leader* as a station that is operational until the end of the execution or a non-faulty station, and was elected as a leader during some execution of procedure Elect-Leader.

Lemma 7. *The total number of rounds during which procedure* Elect-Leader *is run (possibly in several executions) until electing a sustainable leader is* $\log p\frac{4p}{p-f}$ *with probability at least* $1-\frac{1}{p}$.

Proof. Recall that procedure Elect-Leader could be called several times, until selecting a sustainable leader at the latest. The expected number of rounds needed to elect a sustainable leader during these calls is upper bounded by the time needed to hit the first non-faulty station by the executions of procedure Elect-Leader.

We have p stations from which f are prone to crashes. Thus, we have $p-f$ non-faulty stations. That is why the probability that a non-faulty one will respond in the election procedure is at least $(p-f)/p$. We may observe that this probability will increase if we failed in previous executions. In fact, after f executions we may be sure to choose a non-faulty leader. However, we will estimate the probability of our process by an event of awaiting the first success in a number of trials, as our process is stochastically dominated by such a geometric distribution process.

We have a channel without collision detection, so exactly one station has to transmit in order to elect a leader. Let x be the actual number of operational stations. The probability

s of the event that a non-faulty station will be elected in the procedure may be estimated as follows:

$$\begin{aligned}\frac{p-f}{p}\left(1-\frac{1}{p}\right)^{x-1} &\geq \frac{p-f}{p}\left(1-\frac{1}{p}\right)^{p-1}\\ &\geq \frac{p-f}{p}\cdot\frac{1}{4}\cdot\frac{p}{p-1}\\ &= \frac{p-f}{4(p-1)} \geq \frac{p-f}{4p}.\end{aligned}$$

Let us estimate the probability of awaiting the first success in a number of trials. Let $X \sim \mathit{Geom}((p-f)/4p)$. We know that for a geometric random variable with the probability of success equal s:

$$\mathbb{P}(X \geq i) = (1-s)^{i-1}.$$

Applying this to our case with $i = \frac{4p}{p-f}\log p + 1$ we have that

$$\begin{aligned}&\mathbb{P}\left(X \geq \frac{4p}{p-f}\log p + 1\right)\\ &= \left(1-\frac{1}{\frac{4p}{p-f}}\right)^{\frac{4p}{p-f}\log p} \leq e^{-\log p} = \frac{1}{p}.\end{aligned}$$

Thus the probability of a complementary event is

$$\mathbb{P}\left(X < \frac{4p}{p-f}\log p + 1\right) > 1 - \frac{1}{p}.$$

□

Theorem 3. GruTEch *solves Do-All in the channel without collision detection with* $\mathcal{O}(t+p\sqrt{t}+p\ \min\{p/(p-f),t\}\log p)$ *expected work against the Weakly-Adaptive* f*-Bounded adversary.*

Proof. We may divide the work of GruTEch to three components: productive, failing and the one resulting from electing the leader.

Firstly, the core of our algorithm is the same as Groups-Together with the difference that we have the Crash-Echo procedure that takes twice as many transmission rounds.

According to Fact 5, it is sufficient to estimate its work as $\mathcal{O}(t + p\sqrt{t})$.

Secondly, there is some work that results from electing the leader. According to Lemma 7, a sustainable leader will be chosen within $\frac{4p}{p-f}\log p$ rounds of Elect-Leader with high probability. That is why the expected overall work to elect a non-faulty leader is $\mathcal{O}(p\,\frac{p}{p-f}\log p)$.

Finally, there is some amount of failing work that results from rounds where the Crash-Echo procedure indicated that the leader was crashed. However, work accrued during such rounds will not exceed the amount of work resulting from electing the leader, hence we state that failing work contributes $\mathcal{O}(p\,\frac{p}{p-f}\log p)$ as well.

Consequently, we may estimate the expected work of GruTEch as

$$\left(1-\frac{1}{p}\right)\mathcal{O}(t + p\sqrt{t} + p\ \min\{p/(p-f), t\}\log p) + \frac{1}{p}\mathcal{O}(p^2)$$
$$= \mathcal{O}(t + p\sqrt{t} + p\ \min\{p/(p-f), t\}\log p)$$

□

3.3 How GruTEch works for other partial orders

The line of investigation initiated by ROBAL and GruTEch leads to a natural question whether considering some intermediate partial orders of the adversary may provide different work complexities. In this section we answer this question affirmative by examining the GruTEch algorithm against the k-Chain-Ordered adversary on a channel without collision detection.

3.3.1 Lower bound

Theorem 4. *For any reliable randomized algorithm solving Do-All on the multiple-access channel without collision detection and any integer $0 < k \leq f$, there is a k-chain-based partial order of f elements such that the ordered adversary restricted by this order can force the algorithm to perform $\Omega(t + p\sqrt{t} + p\min\{k, f/(p-f), t\})$ expected work.*

Proof. Part $\Omega(t + p\sqrt{t})$ follows from the absolute lower bound on reliable algorithms on a multiple-access channel. We prove the remaining part of the formula. If $k > c \cdot f/(p-f)$, for some constant $0 < c < 1$, then that part is asymptotically dominated by $p\min\{f/(p-f), t\}$ and it is enough to take the order being an anti-chain of f elements. It is a k-chain-based partial order of f elements, and the adversary restricted by this order is equivalent to the Weakly-Adaptive adversary, for which the lower bound $\Omega(p \min\{f/(p-f), t\})$ follows directly from Theorem 2. Therefore, in the remainder of the proof we assume $k \leq c \cdot f/(p-f)$.

Consider the following strategy of the adversary in the first τ rounds, for some value τ to be specified later. Each station which wants to broadcast alone in a round is crashed in the beginning of this round, just before its intended transmission. Let $\mathcal{F}$ be the family of all subsets of stations containing $k/2$ elements. Let $\mathcal{M}$ denote the family of all partial orders consisting of k independent chains of rounded f/k elements each. Consider the first $\tau = k/2$ rounds. For a given $F \in \mathcal{F}$, let the probability of F be an occurrence of an execution during the experiment, in which exactly the stations from set F are failed by round τ. Consider an order M selected uniformly at random from $\mathcal{M}$. The probability that all elements of set $F \in \mathcal{F}$ are in M is a non-zero constant. The result from the following three observations. First, under our assumption, $k < f$ (as $k \leq c \cdot f/(p-f)$ for some $0 < c < 1$). Second, from the proof of the lower bound in [31] with respect to sets of size $\mathcal{O}(f)$, the probability is a non-zero constant provided that in each round we have at most $c' \cdot f$ crashed processes, for some constant $0 < c' < 1$. Third, since each successful station can enforce the adversary to fail at most one chain, after each of the first $\tau = k/2$ rounds there are still at least $k/2$ chains without any failures, hence at most $f/2$ crashes have been enforced and the argument from the lower bound in [31] could be applied. To conclude the proof, non-zero probability of not hitting any element not in M means that there is such $M \in \mathcal{M}$ such that the algorithm does not finish before round τ with constant probability, thus imposing expected work $\Omega(pk)$. □

3.3.2 GruTEch against the k-Chain-Ordered adversary

The analysis of GruTEch against the Weakly-Adaptive adversary relied on electing a leader. Precisely, as we knew that there are $p - f$ non-faulty stations in an execution, then we expected to elect a non-faulty leader in a certain number of trials.

Nevertheless, we could have chosen a faulty station as a leader and the adversary could have chosen to crash that station. However, the number of such failing occurrences would not exceed the number of trials needed to elect the non-faulty one. While considering the k-Chain-Ordered adversary, these estimates are different.

When a leader is elected, then he may belong to the non-faulty set (and this is expected to happen within a certain number of trials) or he may be elected from the faulty set. All stations execute the same code, hence, due to symmetry, we expect that the elected leader is placed somewhere in the adversary's partial order. In what follows we may expect that if the adversary decides to crash the leader, then it will be forced to crash several stations preceding the leader in one of the chains in his partial order. Consequently, this is the key reason why the expected work complexity would reduce against the k-Chain-Ordered adversary.

Theorem 5. GruTEch *solves Do-All in the channel without collision detection with*

$$\mathcal{O}(t + p\sqrt{t} + p\ \min\{p/(p-f), k, t\} \log p)$$

expected work against the k-Chain-Ordered adversary.

Proof. Because of the same arguments as in Theorem 3, it is expected that a non-faulty leader will be chosen in the expected number of $\mathcal{O}(\frac{p}{p-f} \log p)$ trials, generating $\mathcal{O}(p\ \frac{p}{p-f} \log p)$ work.

On the contrary, let us consider what is the amount of work accrued in phases when the leader is chosen from the faulty set and hence may be crashed by the adversary. According to the adversary's partial order we have initially k chains, where chain j has length l_j. If the leader was chosen from that order then it belongs to one of the chains. We will show that it is expected that the chosen leader will be placed somewhere in the middle of that chain.

Let X_j be a random variable representing the position of leader j in a chain. We have that $\mathbb{E}X_j = \sum_{i=1}^{l_j} \frac{i}{l_j} = \frac{1}{l_j} \frac{(1+l_j)}{2} l_j = \frac{(1+l_j)}{2}$.

We can see that if the leader was crashed, this implies that half of the stations forming the chain were also crashed. If at some other points of time, faulty leaders will also be chosen from the same chain, then by simple induction we may conclude that this chain is expected to be all crashed after $\mathcal{O}(\log p)$ iterations, as a single chain has length $\mathcal{O}(p)$ at most. In what follows if there are k chains, then after $\mathcal{O}(k \log p)$ steps this process will end and we may be sure to choose a leader from the non-faulty subset, because the adversary will spend all his failure possibilities.

Finally, if we have a well serving non-faulty leader then the work accrued is asymptotically the same as in Groups-Together algorithm with the difference that each step is now simulated by the Crash-Echo procedure. This work is equal $\mathcal{O}(t + p\sqrt{t})$.

Altogether, taking Lemma 7 into consideration, the expected work performance of

GruTEch against the k-Chain-Ordered adversary is

$$\left(1-\frac{1}{p}\right)\mathcal{O}(t+p\sqrt{t}+p\ \min\{p/(p-f),k,t\}\log p)+\frac{1}{p}\mathcal{O}(p^2)$$

$$=\mathcal{O}(t+p\sqrt{t}+p\ \min\{p/(p-f),k,t\}\log p),$$

what ends the proof.

□

3.3.3 GruTEch against the adversary limited by an arbitrary order

Finally, let us consider the adversary that is limited by an **arbitrary** partial order relation $\succ$ on the set of faulty processors. We say that two partially ordered elements are *incomparable* if none of relations $x \succ y$ and $y \succ x$ hold. Translating this into the considered model, this means that the adversary may crash incomparable elements in any sequence during the execution of the algorithm (clearly, only if x and y are among f stations chosen to be crash-prone).

Theorem 6. GruTEch *solves Do-All in the channel without collision detection with*

$$\mathcal{O}(t+p\sqrt{t}+p\ \min\{p/(p-f),k,t\}\log p)$$

expected work against the k-Ordered adversary.

Proof. We assume that crashes forced by the adversary are constrained by some partial order $\succ$. Let us first recall the following lemma.

Lemma 8. *(Dilworth's Theorem [39]) In a finite partial order, the size of a maximum anti-chain is equal to the minimum number of chains needed to cover all elements of the partial order.*

Recall that the k-Ordered adversary is constrained by an arbitrary order of thickness k. Clearly, the adversary, by choosing some f stations to be crashed, cannot increase the size of the maximal anti-chain. Thus, using Lemma 8 we consider the coverage of the crash-prone stations by at most k disjoint chains, and any dependencies between chains' elements create additional constraints to the adversary, compared to the k-Chain-Ordered one. Hence, we fall into the case concluded in Theorem 5, what completes the proof. □

3.4 GILET — Groups with Internal Leader Election Together

In this section we introduce an algorithm for the channel without collision detection that is designed to work efficiently against the 1-RD adversary. Its expected work complexity is $\mathcal{O}(t + p\sqrt{t}\log^2(p))$. The algorithm makes use of previously designed solutions from [31] i.e., Groups-Together algorithm (cf., Section 2.3.2), yet we implement a major change in how the stations confirm their work (due to the lack of collision detection in the model).

Algorithm 10: GILET; pseudo-code for processor v;

```
arrange all $p$ names of stations into list GROUPS of groups;
initialize variable k := $p/\min\{\lceil\sqrt{t}\rceil, p\}$;
initialize both TASKS and OUTSTANDING$_v$ to sorted list of all $t$ names of tasks;
initialize DONE$_v$ to an empty list of tasks;
initialize REMOVED to an empty list of stations;
repeat
    Epoch-Groups-CW($k$);
until halted;
```

We consider a channel without collision detection. That is why whenever some group g is scheduled to broadcast, a leader election procedure Mod-Confirm-Work is executed in order to hear a successful transmission of exactly one station. Because all the stations within g had the same tasks assigned, then if the leader is chosen, we know that the group performed appropriate tasks.

Such randomized approach to broadcasting progress makes it impossible for the 1-RD adversary to interrupt efficiently, because even if it determines who becomes the leader within a single group, then it cannot crash the leader in the same round in which the broadcast takes place. Hence, the only option for extending the algorithm execution is to crash the whole group. However, crashing whole groups makes that the adversary spends all its crash possibilities faster. Furthermore, the more stations are crashed, the less work generated in the remainder of the algorithm execution.

The inherent cost of such an approach of confirming work is that we may not be sure whether removed groups did really crash. The effect is that if all the tasks were not performed and all the stations were found crashed, then we have to execute an additional procedure that will finish performing them reliably.

This is realized by a new list REMOVED containing removed stations, and procedure

Algorithm 11: EPOCH-GROUPS-CW; pseudo-code for processor v;

```
Input: $k$
set pointer Task_To_Do$_v$ on list TASKS to the initial position of the range $v$;
set pointer Transmit to the first item on list GROUPS;
repeat
    perform the first task on list TASKS, starting from the one pointed to by
     Task_To_Do$_v$, that is in list OUTSTANDING$_v$;
    move the performed task from list OUTSTANDING$_v$ to list DONE$_v$;
    advance pointer Task_To_Do$_v$ by one position on list TASKS;
    if Transmit points to $v$ then
        initialize $i := 0$ ;
        repeat
            execute MOD-CONFIRM-WORK($k$) ;
            if a broadcast was heard then
                break;
            end
            else
                increment $i$ by 1;
            end
        until $i < 4 \log p$;
    end
    if a broadcast was heard in the preceding round then
        let $w$ be the first station in the group pointed to by Transmit;
        for each item $x$ on list DONE$_w$ do
            if $x$ is on list OUTSTANDING$_v$ then
                move $x$ from OUTSTANDING$_v$ to DONE$_v$;
            end
            if $x$ is on list TASKS then
                remove $x$ from TASKS;
            end
        end
        if list TASKS is empty then
            halt;
        end
        advance pointer Transmit by one position on list GROUPS;
    end
    else
        add all the stations from group pointed to by Transmit to list REMOVED;
        remove the group pointed to by Transmit from list GROUPS;
        execute CHECK-OUTSTANDING;
        halt;
    end
until pointer Transmit points to the first entry on list GROUPS;
rearrange all stations in the groups of list GROUPS into a new version of list GROUPS;
```

Algorithm 12: MOD-CONFIRM-WORK; pseudo-code for processor v

```
Input: $k$
$j := 0$;
repeat
    coin $:= \frac{k}{2^j}$;
    toss a coin with the probability coin$^{-1}$ of heads to come up;
    if heads came up in the previous step then
        broadcast $v$ via the channel and attempt to receive a message;
    end
    if some station $w$ was heard then
        break;
    end
    increment $j$ by 1;
until $j < \log k$;
```

Algorithm 13: CHECK-OUTSTANDING; pseudo-code for processor v

```
basing on list REMOVED and list TASKS assign every task to all the processors;
$i := 0$;
repeat
    perform $i$-th task from list TASKS;
    $i := i + 1$;
until $i < |TASKS|$;
clear list TASKS;
```

Check-Outstanding which assigns every remaining task to all the stations. Therefore, if, with some probability, we have mistakenly removed some operational stations, the algorithm still remains reliable.

3.4.1 Analysis of GILET

Lemma 9. GILET *is reliable.*

Proof. As well as in case of GruTEch, the solution does depend on the reliability of Groups-Together, because procedure Mod-Confirm-Work always terminates. If some operational station has been removed from list `GROUPS`, then we execute procedure Check-Outstanding that will finish all the remaining tasks. □

Lemma 10. *Assume that the number of operational stations within a group is in interval* $(\frac{k}{2^{i+1}}, \frac{k}{2^i}]$ *interval and the* `coin` *parameter is set to* $\frac{k}{2^i}$*. Then, during* Mod-Confirm-Work*, a confirming-work broadcast will be performed with probability at least* $1 - \frac{1}{p}$*.*

Proof. We assume that the number of operational stations is in interval $(\frac{k}{2^{i+1}}, \frac{k}{2^i}]$. The probability that exactly one station broadcasts, estimated from the worst case point of view, where only $\frac{k}{2^{i+1}}$ stations are operational, is $\frac{1}{2\sqrt{e}}$, because of the same reason as in Claim 1 of Lemma 4.

That is why we investigate the first success occurrence in a number of trials with the probability of success equal to $\frac{1}{2\sqrt{e}}$.

Let $X \sim Geom\left(\frac{1}{2\sqrt{e}}\right)$. We know that for a geometric random variable with the probability of success equal to s:

$$\mathbb{P}(X \geq i) = (1-s)^{i-1}.$$

We apply it for $i = 2\sqrt{e}\log p + 1$, having that

$$\mathbb{P}(X \geq 2\sqrt{e}\log p + 1) = \left(1 - \frac{1}{2\sqrt{e}}\right)^{2\sqrt{e}\log p} \leq e^{-\log p} = \frac{1}{p}.$$

Thus,

$$\mathbb{P}(X > 2\sqrt{e}\log p + 1) > 1 - \frac{1}{p}.$$

□

Theorem 7. GILET *performs* $\mathcal{O}(t+p\sqrt{t}\log^2(p))$ *expected work on channel without collision detection against the* 1*-RD adversary.*

Proof. The proof of Groups-Together work performance from [31] stated that noisy sparse epochs contribute $\mathcal{O}(t)$ to work and silent sparse epochs contribute $\mathcal{O}(p\sqrt{t})$. Dense epochs also contribute $\mathcal{O}(p\sqrt{t})$ work. Let us compare this with our solution.

Noisy sparse epochs contribute $\mathcal{O}(t)$ because these are phases with successful broadcasts. And there are clearly t tasks to perform, so at most t transmissions will be necessary for this purpose.

Silent sparse epochs, as well as, dense epochs consist of mixed work: effective and failing. In our case, each attempt of transmitting is now simulated by $\mathcal{O}(log^2(p))$ rounds. That is why the amount of work is asymptotically multiplied by this factor. Hence, we have work accrued during silent sparse and dense epochs $\mathcal{O}(p\sqrt{t}\log^2(p))$.

However, according to Lemma 10 with some small probability we could have mistakenly removed a group of stations from list `GROUPS` because Mod-Confirm-Work was silent. Eventually the list of groups may be empty, and there are still some remaining tasks. For such case we execute Check-Outstanding, where all the stations have the same remaining tasks assigned, and do them for |`TASKS`| phases (which actually means until they are all done). It is clear that always at least one station remains operational so all the tasks will be performed. Work contributed in such case is at most $\mathcal{O}(pt)$.

Let us now estimate the expected work:

$$\begin{aligned}\left(1-\frac{1}{p}\right)\mathcal{O}(t+p\sqrt{t}\log^2(p))+\frac{1}{p}\mathcal{O}(pt)\\ =\mathcal{O}(t+p\sqrt{t}\log^2(p)),\end{aligned}$$

what completes the proof. □

3.5 Transition to the beeping model

Up to this point we considered a communication model based on a shared channel, with restriction that collision detection is not available. In this section we consider the beeping model.

It differs from the channel with collision detection by providing slightly different feedback, but we show that Do-All can be solved in the beeping model with the same work complexity.

More precisely, we show that the feedback provided by the beeping channel allows to execute algorithm Groups-Together (cf., Section 2.3.2) and that it is work-optimal as well.

3.5.1 Lower bound

We state the lower bound for Do-All in the beeping model in the following lemma.

Lemma 11. *A reliable algorithm, possibly randomized, with the beeping communication model performs work $\Omega(t + p\sqrt{t})$ in an execution in which no failures occur.*

Proof. The proof is an adaptation of the proof of Lemma 1 from [31] to the beeping model. Let $\mathcal{A}$ be a reliable algorithm. The part $\Omega(t)$ of the bound follows from the fact that every task has to be performed at least once in any execution of $\mathcal{A}$.

Task α is *confirmed* at round i of an execution of algorithm $\mathcal{A}$, if either a station performs a beep successfully and it has performed α by round i, or at least two stations performed a beep simultaneously and all of them have performed task α by round i of the execution. All of the stations broadcasting at round i and confirming α have performed it by then, so at most i tasks can be confirmed at round i. Let $\mathcal{E}_1$ be an execution of the algorithm when no failures occur. Let station v come to a halt at some round j in $\mathcal{E}_1$.

Claim: Tasks not confirmed by round j were performed by v itself in $\mathcal{E}_1$.

Proof of the Claim. Suppose, to the contrary, that this is not the case, and let β be such a task. Consider an execution, say $\mathcal{E}_2$, obtained by running the algorithm and crashing any station that performed task β in $\mathcal{E}_1$ just before it was to perform β in $\mathcal{E}_1$, and all the remaining stations, except for v, crashed at step j. The broadcasts on the channel are the same during the first j rounds in $\mathcal{E}_1$ and $\mathcal{E}_2$. Hence, all the stations perform the same tasks in $\mathcal{E}_1$ and $\mathcal{E}_2$ until round j. The definition of $\mathcal{E}_2$ is consistent with the power of the Unbounded adversary. The algorithm is not reliable because task β is not performed in $\mathcal{E}_2$ and station v is operational. This justifies the claim. □

We estimate the contribution of station v to work. The total number of tasks confirmed in $\mathcal{E}_1$ is at most

$$1 + 2 + \ldots + j = \mathcal{O}(j^2) \ .$$

Suppose some t' tasks have been confirmed by round j. The remaining $t - t'$ tasks have been performed by v. The work of v is at least

$$\Omega(\sqrt{t'} + (t - t')) = \Omega(\sqrt{t}) ,$$

what completes the proof. □

3.5.2 How algorithm Groups-Together works in the beeping model

Collision detection was a significant part of algorithm Groups-Together as it provided the possibility of taking advantage of simultaneous transmissions. Because of maintaining common knowledge about the tasks assigned to groups of stations we were not interested in the content of the transmission but the fact that at least one station from the group remained operational, as this guaranteed progress.

In the beeping model we cannot distinguish between *Single* and *Collision*, however in the sense of detecting progress the feedback is consistent. It means that if a group g is scheduled to broadcast at some phase i, then we have two possibilities. If *Silence* was heard this means that all the stations in group g were crashed, and their tasks remain outstanding. Otherwise, if a beep is heard this means that at least one station in the group remained operational. As the transmission was scheduled in phase i this means that certain i tasks were performed by group g.

Lemma 11 together with the work performance of Groups-Together allows us to conclude that the solution is also optimal in the beeping model.

Corollary 1. Groups-Together *is work optimal in the beeping channel against the f-Bounded adversary.*

3.6 Conclusions

This chapter addressed the challenge of performing work on a shared channel with crash-prone stations against ordered and delayed adversaries, introduced in this work. The considered model is very basic, therefore our solutions could be efficiently implemented in other related communication models with contention and failures.

We found that some orders of crash events are more demanding than others for a given algorithm and the whole problem. In particular, thicker orders or even small delays in the effects of the adversary's decisions allow solutions to stay closer to the absolute lower bound for this problem.

All our algorithms work on a shared channel with acknowledgements only, without collision detection, making the setup challenging. While it was already shown that there is not much we can do against a Strongly-Adaptive f-Bounded adversary, our goal was to investigate whether there are some other adversaries that an algorithm can play against efficiently.

Taking a closer look at our algorithms, each of them works differently against different types of adversaries. ROBAL does not simulate a collision detection mechanism, opposed to the other two solutions, but tries to exploit good properties of an existing (unknown to the algorithm) linear order of crashes. On the other hand, its execution against a Weakly-Adaptive Linearly-Bounded adversary could be inefficient. The adversary, whose crashes are not restricted by any order could enforce a significant increase in the overall work performance by crashing a small number of stations multiple times e.g., just the leaders, and hence Mix-And-Test procedure could generate excessive work by being executed more than $\log p$ times (as is takes place for the Linearly-Ordered f-Bounded adversary). GruTEch, on the other hand, cannot work efficiently against the 1-RD adversary, as there is a global leader chosen to coordinate the Crash-Echo procedure, which simulates confirmations in a way similar to a collision detection mechanism. Hence such an adversary could decide to always crash the leader, making the algorithm inefficient, as electing a leader is quite effortful — the leader is chosen in a number of trials, what generates excessive work. On the other hand, GILET confirms every piece of progress by electing a leader in a specific way, what proved to be efficient against the 1-RD adversary, but executing it against the Weakly-Adaptive adversary would result in an increase in the overall work complexity.

Remarks on time complexity. First of all, we emphasize that time complexity, defined as the number of rounds until all non-crashed stations terminate, is not the best choice to describe how efficient the algorithms are, because this strongly depends on how the adversary interferes with the system. In what follows, we present some general bounds that might, however, overestimate the time complexity for a vast range of executions.

In all our considerations, at some point of an execution (even at the very beginning) it may happen that only non-faulty stations remain operational, because the adversary will use all of its possible crashes. Then at most t tasks must be performed by the remaining $p - f$ stations. Hence, even if the tasks are equally distributed among non-faulty stations, doing them all lasts at least $t/(p-f)$ rounds. On the other hand, initially t tasks are distributed among p stations. Thus, on average, a station will be working on t/p tasks. If now the adversary decides to crash a station just before it was to confirm its tasks, then this prolongs the overall execution by t/p rounds. Because there are f crashes, then at most tf/p rounds are additionally needed to finish. However, it is also true that stations are capable of performing t tasks in $\sqrt{t}$ rounds. This corresponds to the triangular Two-Lists-fashion of assigning tasks to stations cf., Figure 2.1 In this view each crash enforces an additional step of the execution, what gives us the upper bound of around $f+\sqrt{t}$ rounds.

All our algorithms undergo the same time bounds for actually performing tasks or suffering crashes as mentioned above. Additionally, $\mathcal{O}(\sqrt{t}\log p)$ rounds are needed for ROBAL to select sets of leaders throughout all the executions of the Mix-And-Test procedure. Consequently, the expected running time of ROBAL is $\mathcal{O}\left(\frac{t}{p-f}+\min\left\{\frac{tf}{p}, f+\sqrt{t}\right\}+\sqrt{t}\log p\right)$. Following the same reasoning, GruTEch algorithm, apart from doing productive work in presence of the adversary, requires additional time for the leader election mode, which is $\mathcal{O}\left(\frac{p}{p-f}\log p\right)$ in expectation. The total expected running time of GruTEch is therefore $\mathcal{O}\left(\frac{t}{p-f}+\min\left\{\frac{tf}{p}, f+\sqrt{t}\right\}+\frac{p}{p-f}\log p\right)$. In GILET choosing the leader confirms performing tasks, hence its expected running time is $\mathcal{O}\left(\left(\frac{t}{p-f}+\min\left\{\frac{tf}{p}, f+\sqrt{t}\right\}\right)\log^2(p)\right)$.

Remarks on energy complexity. Since our algorithms are randomized, it seems to be quite difficult to state tight bounds for the transmission energy used in executions. Here by transmission energy we understand the total number of transmissions undertaken by stations during the execution. Thus, transmission energy is also equivalent to message complexity. Nevertheless, assuming that n denotes the number of operational stations and there is a certain amount of work S accrued until a certain point of time of an execution of any of our algorithms, then $S/\sqrt{n}$ is, roughly, the upper bound on the number of transmissions performed by that time. This is because substantial parts of our algorithms are based on procedure Groups-Together, in which roughly $\sqrt{n'}$ stations in a group transmit in a round, out of at least $n' \geq n$ operational ones that contribute to the total work S. However, our algorithms also strongly rely on different leader election procedures, therefore the total transmission energy cost in an execution may vary significantly.

Open problems. Further study of distributed problems and systems against ordered adversaries seems to be a natural future direction. Another interesting area is to study various extensions of the Do-All problem in the shared-channel setting, such as considering a dynamic model, where additional tasks may appear during an algorithm execution or considering tasks with deadlines. In other words, to develop a scheduling theory on a shared channel prone to crash-failures. It is also interesting to consider a centralized approach for scheduling tasks and compare it with the distributed approach presented in this chapter. This means, in particular, to examine whether centralized scheduling does not violate the assumptions for the considered model, and whether it could give better bounds in terms of work. Furthermore, considering algorithms for the Do-All problem on a multiple-access channel against adversaries, whose decisions are fully random could complement the hierarchy of adversaries presented in this chapter. In all the above mentioned directions, including the one considered in this work, one of the most fundamental questions arises: is there a universal, efficient solution against the whole range of adversarial scenarios? Different properties of adversaries and the corresponding algorithms discussed above suggest that it may be difficult to design such a universally efficient algorithm.

Chapter 4

Performing arbitrary length tasks

In this chapter, we consider fault-tolerant scheduling of t arbitrary length tasks on p processors reliably over a multiple-access channel without collision detection. The contribution of this chapter is threefold, on: deterministic preemptive, deterministic non-preemptive, and randomized preemptive settings. In the setting with preemption, we prove a lower bound $\Omega(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$ on work, where L is the sum of lengths of all tasks and α is the maximum length of a task, which holds for deterministic and randomized algorithms against a Strongly-Adaptive adversary. We design a deterministic algorithm, called ScaTri, achieving work $\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$, which matches the lower bound asymptotically.

In the model without task preemption, we show a slightly higher lower bound on work $\Omega\left(L + \frac{L}{t}p\sqrt{t} + p\min\{f, t\} + p\alpha\right)$, implying a separation between the two settings: with and without preemption. Similarly as in the previous setting, it holds for deterministic and randomized algorithms against the Strongly-Adaptive f-Bounded adversary. We develop a deterministic algorithm, called DefTri, achieving work $\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p\min\{f, t\})$.

algorithm	**work**	**ref.**	**lower bound**	**ref.**
ScaTri	$\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$	Sec 4.1.2 Thm 9	$\Omega(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$	Sec 4.1.1 Thm 8
DefTri	$\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p\min\{f, t\})$	Sec 4.2.2 Thm 10	$\Omega\left(L + \frac{L}{t}p\sqrt{t} + p\min\{f, t\} + p\alpha\right)$	Sec 4.2.1 Cor 3
RanScaTri	$\mathcal{O}(L + p\sqrt{L}\log p + p\alpha)$	Sec 4.3 Thm 11	$\Omega(L + p\sqrt{L} + p\alpha)$	Sec 4.1.1 Lem 13

Table 4.1: Summary of main results in Chapter 4. ScaTri and DefTri are deterministic algorithms analyzed against the Strongly-Adaptive f-Bounded adversary, while RanScaTri is a randomized one and its work performance is in expectation. It is analyzed against the Non-Adaptive f-Bounded adversary. All solutions work on a channel without collision detection.

We then investigate the scenario with randomization and derive a randomized distributed preemptive algorithm, called RanScaTri, against Non-Adaptive adversaries. This is a Las Vegas algorithm which achieves expected work $\mathcal{O}(L + p\sqrt{L}\log p + p\alpha)$, and thus shows that randomization helps against the Non-Adaptive adversary, increasing the performance of the algorithm close to the lower bound $\Omega(L + p\sqrt{L} + p\alpha)$ (see Lemma 13). Results are discussed and compared in detail in Section 4.4. Figure 4.1 illustrates the complexity for different variants of the considered problem.

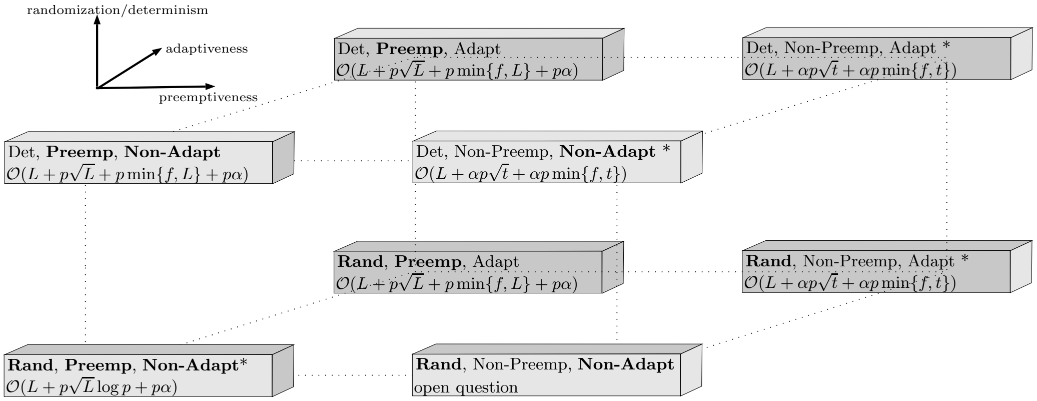
randomization/determinism
adaptiveness
preemptiveness
Det, **Preemp**, Adapt
$\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$
Det, Non-Preemp, Adapt *
$\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p\min\{f, t\})$
Det, **Preemp**, **Non-Adapt**
$\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$
Det, Non-Preemp, **Non-Adapt** *
$\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p\min\{f, t\})$
Rand, **Preemp**, Adapt
$\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$
Rand, Non-Preemp, Adapt *
$\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p\min\{f, t\})$
Rand, **Preemp**, **Non-Adapt***
$\mathcal{O}(L + p\sqrt{L}\log p + p\alpha)$
Rand, Non-Preemp, **Non-Adapt**
open question

Figure 4.1: Illustration of the complexity of scheduling on a MAC. Legend: Rand - randomized algorithm, Det - deterministic algorithm, Preemp - preemptive setting, Non-Preemp - non-preemptive setting, Adapt - adaptive adversary, Non-Adapt - non-adaptive adversary. We consider three different features depending on whether they are present or not: the preemptiveness feature changes along the X-axis, adaptiveness of the adversary changes along the Z-axis, and randomization/determinism changes along the Y-axis. Features written in bold denote an easier configuration. Boxes with * indicate that the presented solution for such configuration has a gap when confronted with the lower bounds that we present. Results on the front facet (lighter boxes) are for the non-adaptive adversary indicating that (i) preemption improves deterministic algorithms and (ii) randomization improves preemptive algorithms. Results on the rear facet (darker boxes) are for the adaptive adversary, indicating that preemption improves both deterministic and randomized algorithms.

4.1 Preemptive model

In this section, we consider the scheduling problem in the model with preemption, which is, intuitively, an easier model to tackle. As mentioned before, the multiple-access channel has no collision detection. Firstly, we show a lower bound for oblivious tasks (Section 4.1.1) and then present and analyze our algorithm for non-oblivious tasks (Section 4.1.2). Notice that a lower bound for oblivious tasks is a stronger lower bound as it also applies for non-oblivious tasks and similarly an upper bound for non-oblivious tasks is a stronger upper bound. Since both of our bounds are matching, this implies that with preemption, the distinction between oblivious tasks and non-oblivious tasks does not matter.

4.1.1 Lower bound

We once again recall the minimal work complexity for the Do-All problem on a shared channel, introduced in [31].

Lemma 12 ([31], Lemma 2)**.** *A reliable, distributed algorithm, possibly randomized, performs $\Omega(t + p\sqrt{t})$ work in an execution in which no failures occur.*

Recall that L denotes the total length of tasks and α denotes the length of the longest task. As tasks are built with unit length subtasks, we can look at this result from the subtask perspective. Precisely, as L can be considered to represent the number of subtasks needed to be performed by the system, then the lower bound for our model translates in a straightforward way. Furthermore, in our model there is a certain bottleneck dictated by the longest task. The reason for this is that if there is an execution with just a single long task, then it is the only one that processors can work on.

Lemma 13. *A reliable, distributed algorithm, possibly randomized, performs $\Omega(L+p\sqrt{L}+p\alpha)$ work, even in an execution in which no failures occur on any set of oblivious tasks with arbitrary lengths with preemption.*

Proof. $\Omega(L + p\sqrt{L})$ follows from Lemma 12. It suffices to concentrate on the remaining part.

Let us consider an arbitrary reliable algorithm $\mathcal{A}$. Let $\mathcal{E}$ be an execution of algorithm $\mathcal{A}$ with a set of oblivious tasks $\mathcal{T}$ with preemption. Set $\mathcal{T}$ contains the longest task i, which has length α. Hence, task i consists of α subtasks, which cannot be divided between processors, to be done in parallel. In other words, if we consider some round r, then an arbitrary subtask i_j and subtask i_{j+1} of i cannot be performed in the system in round r, because subtask i_{j+1} can be performed only if subtask i_j was already done. Therefore, task i requires α rounds to be done.

Because we assume that $\mathcal{A}$ is reliable, then no processor halts before task i is performed. To prove this, let us assume, to the contrary, that some processor v halts during $\mathcal{E}$, before task i is performed. After v halts, the $(p-1)$-Bounded adversary (which $\mathcal{A}$, as a reliable algorithm, is also required to handle) crashes all processors but v. This results in i being undone, while v is an operational processor, which cannot restart and perform i. This contradicts the reliability of $\mathcal{A}$.

We conclude that at least α rounds are required to perform all tasks from $\mathcal{T}$ and at most p processors are operational during these rounds, so this generates $\Omega(p\alpha)$ work and ends the proof.

□

In presence of failures, we extend the following result.

Lemma 14 ([31], Theorem 2). *The Strongly-Adaptive f-Bounded adversary, for $0 \leq f < p$, can force any reliable, possibly randomized, algorithm to perform $\Omega(t + p\sqrt{t} + p\min\{f,t\})$ work.*

Combining all the results above, we conclude with the following theorem, setting the lower bound for the considered problem.

Theorem 8. *The Strongly-Adaptive f-Bounded adversary, for $0 \leq f < p$, can force any reliable, possibly randomized and centralized algorithm and a set of oblivious tasks of arbitrary lengths with preemption, to perform $\Omega(L + p\sqrt{L} + p\min\{f,L\} + p\alpha)$ work.*

Proof. The result follows explicitly by combining Lemma 13 and Lemma 14 applied for $t = L$, as in the preemptive model subtasks may be treated as independent tasks. □

4.1.2 Algorithm ScaTri

In this section we present our algorithm Scaling-Triangle MAC Scheduling (ScaTri for short). The most important assumption for ScaTri is that processors have access to all the tasks and the corresponding subtasks that build those tasks. This means that they know all the tasks (and subtasks) and their identifiers, as well as their lengths. Every subtask has its unique identifier, which allows to discover to which task it belongs and what is its position in that task. This allows to preserve consistency and coherency: subtask k cannot be performed before subtask $k-1$.

In what follows we assume that each processor maintains three lists: `STATIONS`, `SUBTASKS` and `TASKS`. The first list is a list of operational processors and is updated according to the information broadcasted through the communication channel. If there is information that some processor was recognized as crashed, then this processor is removed from the list. In the context of the TaPeBB procedure, recall that this is realized as the output information. TaPeBB returns the list of crashed processors so that operational processors may update their lists.

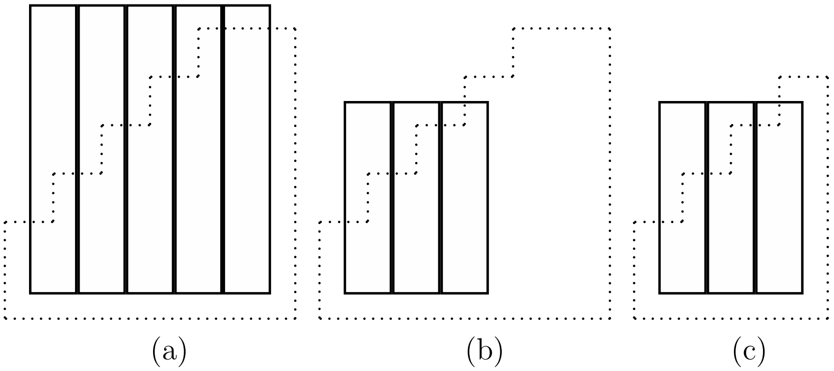

Figure 4.2: A very general idea about how ScaTri works. Vertical stripes represent tasks and the dotted-line triangle is the capability of the algorithm to perform tasks in a single epoch (or parts of them i.e. some consecutive subtasks). Consequently, if there are enough tasks to pack into an epoch, then it is executed (a). Otherwise, if there are not enough tasks for the current length of an epoch (b), then the length of the epoch is reduced (c). This helps preventing excessive idle work.

List `SUBTASKS` represents all the subtasks that are initially computed from the list of tasks. If some subtasks are performed then this fact is also updated on the list. List `TASKS` represents the set of tasks — it is a convenient way to know what are the consecutive parts of input for the TaPeBB procedure. Tasks are assigned to each processor at the beginning of an epoch (TaPeBB execution). We assume that processors have instant access to tasks' lengths. Information on list `TASKS` may be updated directly from information maintained on list `SUBTASKS`. To simplify the discussion, by $|\texttt{XYZ}|$ we denote the length of list `XYZ` and by $P = |\texttt{STATIONS}|$ the actual number of operational processors.

While introducing TaPeBB (cf., Section 2.3.3) we mentioned that the capability of performing subtasks in an epoch is at most $\sum_{j=1}^{d} j$ which is the sum of an arithmetic series with common difference equal 1 over all rounds of an epoch. If we take a geometric approach and draw lines or boxes of increasing lengths one next to the other, then drawing this sequence would form a triangle. Hence, providing a subset of tasks as the input is equivalent to packing them into such a triangle, cf., Figure 4.2.

In TaPeBB executed with $\phi = 1$ broadcasts take place each round. This means that in round 1 processor 1 is scheduled to broadcast, in round 2 processor 2 and, in general, in round j processor j is scheduled to broadcast. Consequently, j subtasks can be confirmed as performed in round j, unless processor j is crashed and hence silence is heard in round j. Therefore, the capability of performing subtasks by TaPeBB is referred to as the triangle.

We assume d is a parameter describing the current length of an epoch. Initially it is set to p, but it may be reduced while the algorithm is running, cf., Figure 4.2 (b), (c). In what follows, let us assume that the length of the epoch d is set to $p/2^i$ for some i. We need to fill in a triangle of size $\sum_{j=1}^{\frac{p}{2^i}} j$. Initially we need to provide $p/2^i$ tasks, that will form the base of the triangle. Tasks are provided as the input in such a way that the shortest ones are preferred for processors with lower id's. If there are several tasks with same lengths, then those with lower id's are preferred, cf., Figure 4.3. After having the base filled, it is necessary to look whether there are any gaps in the triangle, see Figure 4.3 (b). If so, another layer of tasks is placed on top of the base layer, and the procedure is repeated, see Figure 4.3 (c), (d). Otherwise, we are done and ready to execute TaPeBB.

This approach allows to "trim" longer tasks preferably — these will have more subtasks completed after executing TaPeBB than shorter ones, because they are scheduled to be done by processors with higher id's, which are broadcasting in further rounds. As processors with higher id's are broadcasting in further rounds, then they can confirm more subtasks with a single broadcast.

One should observe that performing a transmission is an opportunity to confirm on the channel that the subtasks that were assigned to a processor are done. Additionally this confirms that a certain processor is still operational. In what follows these two types of aggregated pieces of information: which tasks were done, and which processors were crashed, are eventually provided as the output of TaPeBB.

When $d = p/2^i$ for some $i = 1, \ldots, \log p$ it may happen that there are not enough subtasks (tasks) to fill in a maximal triangle. If this happens we will, in some cases, reduce the task-schedule triangle (and simultaneously the length of the epoch) to $p/2^{i+1}$ and try to fill in a smaller triangle, cf., Figure 4.2 (b), (c).

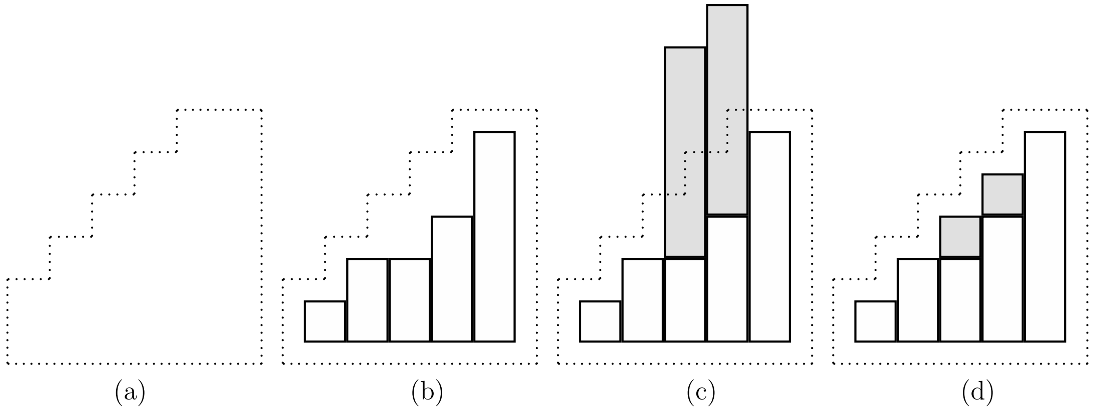

Figure 4.3: An illustration of task input. (a) Initially there is a certain capability of TaPeBB (triangle size) for performing tasks. (b) The initial layer is filled with tasks (white blocks), yet there are still some gaps. (c) An additional layer (gray blocks) is placed to fill in the triangle entirely. (d) In fact those additional tasks will only be done partially.

We emphasize that we described the process of assigning tasks to processors from the algorithm perspective, i.e., we illustrated that the system needs to provide input to TaPeBB. Nevertheless, for the sake of clarity we assume that TaPeBB collects the appropriate input by itself according to the rules described above, after having lists `TASKS` and `STATIONS` provided as the input. Figures 4.2 and 4.3 illustrate the idea standing behind ScaTri.

The following theorem summarizes work performance of ScaTri:

Theorem 9. ScaTri *performs $\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$ work against the adaptive adversary and a set of non-oblivious, arbitrary length tasks with preemption.*

Proof. The main part of ScaTri is designed as a repeat loop. Therefore, the total contribution to work of ScaTri is divided between three types of iterations of that loop, which we analyze separately:

- Case 1: iterations in which $|\texttt{TASKS}| \geq d$ and $|\texttt{SUBTASKS}| \geq d(d+1)/2$

  If the algorithm reaches such a condition, it means that there is a significant number of tasks and subtasks. That is why work accrued in rounds with successful transmissions accounts to L. However, processors which did not crash during the execution of TaPeBB still generate work after their transmissions, so we need to calculate this kind of work as well. Let us call it *active idle work*.

Algorithm 14: ScaTri; pseudo-code for processor v

```
initialize STATIONS to a sorted list of all $p$ names of processors;
initialize TASKS to a sorted list of all $t$ names of tasks;
initialize SUBTASKS to a sorted list of all subtasks according to the information from
  TASKS;
initialize variable $d$ representing the length of an epoch;
initialize variable $\phi := 1$ representing the length of a phase;
initialize $i := 0$;
repeat
    $d := p/2^i$;
    if $|SUBTASKS| \geq d(d+1)/2$ then
        execute TaPeBB($v, d,$ TASKS, STATIONS, $\phi$);
        update TASKS, STATIONS, SUBTASKS according to the output information from
          TaPeBB;
    end
    else
        $i := i + 1$;
    end
until $|TASKS| = 0$;
```

Suppose processor v is scheduled to broadcast during the i'th round of TaPeBB execution. If the broadcast takes place, then processor v generated i units of work, since it performed i subtasks. If we add the active idle work performed in the i'th round by all processors that already performed their broadcasts, then this means that we can assign up to 2 units of work to every subtask performed by a processor which performed the broadcast. Hence the amount of active idle work also accounts to L.

Work in rounds with failing transmissions can be computed as $p \min\{f, L\}$. Precisely, there are clearly f crashes that may occur resulting from the adversary activity and because the length of an epoch is at most p, then the amount of wasted work for each crash is at most p. On the other hand, the number of crashes in the wasted work calculation cannot exceed the overall number of subtasks, hence the minimum factor.

- Case 2: iterations in which $|$TASKS$| < d$ and $|$SUBTASKS$| \geq d(d+1)/2$

  If we fall into such case, there is a significant number of subtasks, but they all form few long tasks. This means that the size of an epoch has to be reduced significantly and those tasks need to be performed by few initial processors in a certain number of epochs.

Let us recall a fact from Section 2.3.3, that when $|\texttt{TASKS}| < d$, then the duration of a TaPeBB epoch is $|\texttt{TASKS}|$. Tasks are always provided to TaPeBB in a sorted way, hence shorter ones are scheduled to processors, which are going to broadcast earlier. The length of the longest task is α, so it is always trimmed by one subtask, excluding crashes (work of which accounts to $p\min\{f, L\}$). In what follows, α rounds suffice to perform all tasks within the considered case.

The magnitude of work for such cases is $\mathcal{O}(p\alpha)$. This is justified by the fact, that only an initial segment of processors perform actual tasks, yet there may be at most p operational processors in the system, some of which generating *passive idle work* (i.e., they did not have any subtasks assigned from the beginning) each round and the number of rounds to deal with such cases is dictated by the longest task of length α.

- Case 3: iterations in which $|\texttt{SUBTASKS}| < d(d+1)/2$

  The algorithm begins with the length of the epoch set as p. When the number of subtasks is relatively small (no matter what is their distribution over tasks) this means that the algorithm should reduce the size of the schedule triangle to $p/2$ in order to reduce the number of passive idle work of processors that do not have any subtasks assigned. Of course the productive part of work after rescaling the triangle, generated by processors that transmitted successfully is accounted to L, so as active idle work, and wasted part of work produces $p\min\{f, L\}$ work.

  We now compute the passive idle work of remaining processors. The ratio between the initial triangle size and the reduced one is

$$\frac{\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}}{\frac{p(p+1)}{2}} = \frac{1}{4}\,\frac{p+2}{p+1} \geq \frac{1}{4}\ ,$$

  hence we conclude that $\mathcal{O}(1)$ epochs is enough for a single case.

  Having in mind that we are computing the overall work in the considered case as a union bound, we examine the inequality $p(p+1)/2 > L$. It states that there are not enough subtasks to fill in the initial triangle — the size of the triangle is therefore reduced to $p/2$ and we assume that such reduced triangle schedule can be filled with tasks entirely i.e., $\frac{p}{2}(\frac{p}{2}+1)/2 \leq L$. Hence, because its size is reduced to $p/2$, it means that now the number of idling rounds within this case is bounded by $p/2 = \mathcal{O}(\sqrt{L})$.

> Together with the fact that the number of operational processors is at most p and that the number of epochs sufficing to perform all the tasks within a single case is constant we compute the total work as follows. Following Fact 6, we have that $\mathcal{O}(1)$ epochs × p processors passively idling at most for $\mathcal{O}(\sqrt{L})$ rounds of an epoch is the work accrued in a single case. However, if it happens that the triangle needs to be reduced once more, then we will have that $p/4 = \mathcal{O}(\sqrt{L}/2)$.
>
> Because there is a logarithmic number of such cases and the reasoning for each of them is consistent with the above mentioned argument (there are not enough tasks to fill in the triangle of size $p/2^i$, hence the size is reduced to $p/2^{i+1}$, for which condition $p/2^{i+1} \leq \sqrt{L}/2^j$ holds, for some $0 \leq j \leq \log p$), basing once again on Fact 6 we compute the total idle work as a union bound of all the cases, having the following:
>
> $$cp \sum_{i=0}^{\log p} \frac{\sqrt{L}}{2^i} = \mathcal{O}(p\sqrt{L}) \ ,$$
>
> where c is the constant number of epochs required to perform everything within a single case.

Note that we consider five types of work: active and passive idling, wasted, the one following from a bottleneck scenario (when there are few very long tasks) and productive. Passive idle work has been bounded by the union of several cases when the length of an epoch is reduced, yet there are still some operational processors that may generate passive idle work.

Wasted work is a result of the adversary and is clearly related to crashes that may occur. However, if some subtask k is scheduled to be done in an epoch by processor v, then no other processor apart from v has subtask k scheduled in this particular epoch. In what follows there are two possibilities: either subtask k is done by v, and the generated productive work is accounted to L (twice for subtask k in the entire execution, because of active idle work), or processor v is crashed and the work accrued by v is categorized as wasted (factor $p\min\{f, L\}$) and k is scheduled once again in the following epoch.

Consequently, considering all the cases we have that ScaTri has $\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$ work complexity against an adaptive adversary. □

As mentioned in the beginning of Section 4.1, the lower bound here is proved for oblivious tasks in the preemptive model, while the algorithm works reliably for non-oblivious tasks in the same model, because TAPEBB procedure provides any intermediate task performing progress as the output information and all the information is updated sequentially after each epoch. This implies that the distinction between oblivious and non-oblivious tasks in the preemptive model and against an adaptive adversary does not matter, thus we finish this section with the following corollary:

Corollary 2. SCATRI *is optimal in asymptotic work efficiency for tasks with arbitrary lengths with preemption for both oblivious and non-oblivious tasks.*

4.2 Non-preemptive model

In this section we consider a complementary problem of performing tasks when preemption is not available, thus every task needs to be performed in one attempt by a single processor in order to be confirmed. Again, we begin with the lower bound and then proceed to the algorithm and its analysis.

4.2.1 Lower bound

The non-preemptive model is characterized by the fact that each task can only be performed by a processor in one piece. This reflects an all-or-nothing policy, i.e., a processor cannot make any intermediate progress while performing a task. If it starts working on a task then either it must be done entirely or it is abandoned without any subtasks actually performed.

In what follows, any intermediate broadcasts while performing a task are not helpful for the system. The only meaningful transmissions are those which allow to confirm certain tasks being done.

We begin with a general lower bound for the model, even if the adversary does not distract the system. For the entire section let us recall that we defined a phase as the time step between consecutive broadcasts.

Lemma 15. *A reliable algorithm, possibly randomized, performs $\Omega(L + \frac{L}{t}p\sqrt{t})$ work in an execution in which no failures occur.*

Proof. It is sufficient to consider the channel with collision detection in which processors, possibly, could have more information. Let $\mathcal{A}$ be a reliable algorithm. The part L of the bound follows from the fact that all the tasks are built from L subtasks, thus all the subtasks need to be done at least once in any execution of $\mathcal{A}$. In other words, there are at least L units of work required to perform all the tasks.

Task a is *confirmed* at phase γ of an execution of algorithm $\mathcal{A}$, if either a processor broadcasts successfully and it has performed a by the end of phase γ, or at least two processors broadcast simultaneously and all of them, with a possible exception of one processor, have performed task a by the end of phase γ of the execution. All of the processors broadcasting at phase γ and confirming task a have performed it by then, so at most 2γ tasks can be confirmed at phase γ.

Let $\mathcal{E}_1$ be an execution of the algorithm when no failures occur. Let v be the processor that comes to a halt at some phase γ' in $\mathcal{E}_1$.

Claim. *Tasks not confirmed by the end of phase γ' were performed by v itself in $\mathcal{E}_1$.*

Proof of the Claim. Suppose, to the contrary, that this is not the case, and let b be such a task. Consider an execution, say $\mathcal{E}_2$, obtained by running the algorithm and crashing any processor that performed task b in $\mathcal{E}_1$ just before the processor finishes performing task b in $\mathcal{E}_1$, and all the remaining processors, except for v, crashed at phase γ'. Broadcasts on the channel are the same during the first γ' phases in $\mathcal{E}_1$ and $\mathcal{E}_2$. Hence all processors perform the same tasks in $\mathcal{E}_1$ and $\mathcal{E}_2$ until the beginning of phase γ'. The definition of $\mathcal{E}_2$ is consistent with the power of the *unbounded* adversary. The algorithm is not reliable because task b is not performed in $\mathcal{E}_2$ and processor v is operational. We obtain a contradiction which justifies the claim. □

We compute the contribution of processor v to work. The total number of tasks confirmed in $\mathcal{E}_1$ is at most

$$2(1+2+\ldots+\gamma') = \mathcal{O}((\gamma')^2) \ .$$

Suppose some t' tasks have been confirmed by the end of phase γ'. The remaining $t-t'$ tasks have been performed by v and each of them required $\frac{L}{t}$ units of work what reflects the average length of a task. The work of v is therefore

$$\Omega\left(\frac{L}{t}\sqrt{t'} + \frac{L}{t}(t-t')\right) = \Omega\left(\frac{L}{t}\sqrt{t}\right) \ ,$$

what completes the proof. □

Lemma 16. *A reliable, distributed algorithm, possibly randomized, performs*

$$\Omega\left(L + \frac{L}{t}p\sqrt{t} + p\min\{f,t\} + p\alpha\right)$$

work in an execution with at most f failures against an adaptive adversary.

Proof. We consider two cases, depending on which term dominates the bound. If it is $\Omega\left(L + \frac{L}{t}p\sqrt{t} + p\alpha\right)$, then the bound follows from combining Lemma 15 with Lemma 13 — if the longest task was a bottleneck in the preemptive model, then, certainly, it is a bottleneck in the non-preemptive model.

Consider the case when $\Omega(p\min\{f,t\})$ determines the magnitude of the bound. Denote $g = \min\{f,t\}$. Let $\mathcal{E}_1$ be an execution obtained by running the algorithm and crashing any processor that wants to broadcast as a single one during the first $g/4$ phases. Denote as A the set of processors failed in $\mathcal{E}_1$. The definition of $\mathcal{E}_1$ is consistent with the power of the Strongly-Adaptive f-Bounded adversary, since $|A| \le g/4 \le f$.

Claim. *No processor halts by phase $g/4$ in execution $\mathcal{E}_1$.*

Proof of the Claim. Suppose, to the contrary, that some processor v halts before phase $g/4$ in $\mathcal{E}_1$. We show that the algorithm is not reliable. To this end we consider another execution, say, $\mathcal{E}_2$ that can be enforced by the Unbounded adversary. Let a be a task which is performed in $\mathcal{E}_1$ by at most

$$\frac{pg}{4(t-g/4)} \le \frac{pg}{3t}$$

processors, except for processor v, during the first $g/4$ phases. It exists because $g \le t$. Let B be this set of processors. Size $|B|$ of set B satisfies the inequality

$$|B| \le \frac{pg}{3t} \le \frac{p}{3} \ .$$

We define operationally a set of processors, denoted C, as follows. Initially C equals $A \cup B$. Notice that the inequality $|A \cup B| \le 7p/12$ holds. If there is any processor that wants to broadcast during the first $g/4$ phases in $\mathcal{E}_1$ as the only processor not in the current C, then it is added to C. At most one processor is added to C for each among the first $g/4 \le p/4$ phases of $\mathcal{E}_1$, so $|C| \le 10p/12 < p$.

Let $\mathcal{E}_2$ be an execution obtained by failing all the processors in C at the start and then running the algorithm. The definition of $\mathcal{E}_2$ is consistent with the power of the Unbounded adversary. There is no broadcast heard in $\mathcal{E}_2$ during the first $g/4$ phases. Therefore each processor operational in $\mathcal{E}_2$ behaves in exactly the same way in both $\mathcal{E}_1$ and $\mathcal{E}_2$ during the first $g/4$ phases. Task a is not performed in execution $\mathcal{E}_2$ by the end of phase $g/4$, because the processors in B have been failed and the remaining ones behave as in $\mathcal{E}_1$.

Processor v is not failed in $\mathcal{E}_2$ and so it performs the same actions in both $\mathcal{E}_1$ and $\mathcal{E}_2$. Consider a new execution, denoted $\mathcal{E}_3$. This execution is like $\mathcal{E}_2$ till the beginning of phase $g/4$, then all the processors, except for v, are failed. The definition of $\mathcal{E}_3$ is consistent with the power of the Unbounded adversary. Processor v is operational but halted and task a is still remaining in $\mathcal{E}_3$ at the end of phase $g/4$. We conclude that the algorithm is not reliable. This contradiction completes the proof of the claim. □

Consider the original execution $\mathcal{E}_1$ again. It follows from the claim that there are at least $p - g/4 = \Omega(p)$ processors still operational and non-halted by the end of phase $g/4$ in execution $\mathcal{E}_1$. Consequently, since the minimal length of a phase is equal 1 (i.e., it is consistent with a round) and all crashes happened one after another in consecutive phases, then processors generated work $\Omega\left(p \cdot \min\{f, t\}\right)$ by the end of phase $g/4$, what ends the proof. □

Corollary 3. *There are non-preemptive task inputs for which a reliable, distributed algorithm, possibly randomized, performs $\Omega\left(L + \frac{L}{t}p\sqrt{t} + p\min\{f,t\} + p\alpha\right)$ work against the Strongly-Adaptive f-Bounded adversary.*

4.2.2 Algorithm DefTri

The key reason to consider the notion of a phase (i.e. time between consecutive broadcasts), introduced in Section 2.3.3, is that it allows to assume broadcasts are done after tasks are done entirely. The only question is how to set the phase parameter to be sure that processors had time to perform tasks before it is their turn to broadcast.

We analyze our algorithm from the average length of the current set of tasks perspective. To justify this setting let us consider two scenarios. For the first one, when the phase parameter is set to 1, which is consistent with transmissions taking place each round, we already mentioned above that most of the transmissions might have no effect on progressing with performing tasks.

However, setting the phase parameter to the length of the longest task α can generate excessive idle work. If each phase lasts α, then for short tasks processors perform the tasks

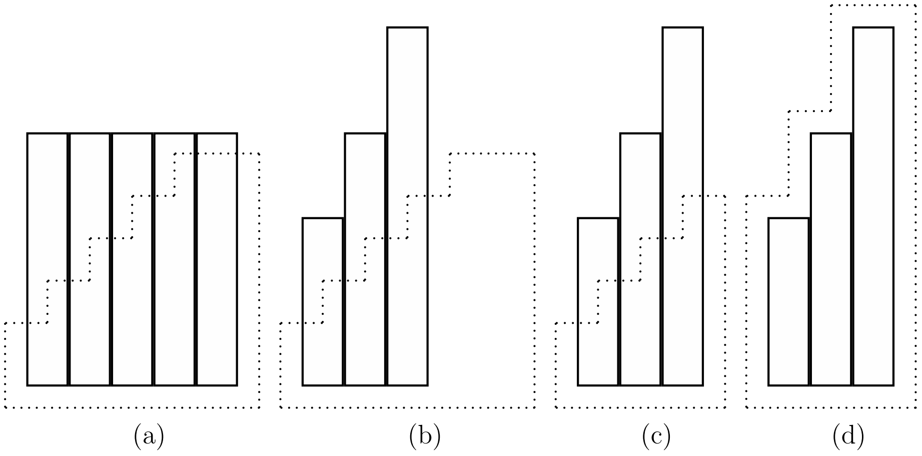

Figure 4.4: Most important features of DefTri. Similarly to ScaTri we apply the method of reducing idle work by reducing the input size (epoch length) for the task performing procedure ((a), (b), (c)). Additionally, we change the duration of phases (time between consecutive broadcasts) in order to be able to fill in tasks entirely - in the model without preemption tasks cannot be done partially (d).

and then stay idle until the broadcast.

In what follows, setting the phase parameter to the average length of the current set of tasks allows us to estimate the number of tasks that can be performed in a certain period. What is more it proves that this setting always allows to schedule a significant number of tasks to be done, while preventing excessive idle work. Hence we begin with a simple fact showing that the number of tasks with length twice the average does not exceed half of the total number of tasks.

Fact 9. *Let t be the number of tasks, L be the sum of all the lengths of tasks and let $\frac{L}{t}$ represent the average length of a task. Then we have that $\left|\left\{a : \ell_a > 2\frac{L}{t}\right\}\right| \leq \frac{t}{2}$.*

Proof. We have that $L = \sum_{i \leq t} \ell_i$, so

$$L = \sum_{i \leq t} \ell_i \geq \sum_{\left\{a:\ell_i > 2\frac{L}{t}\right\}} \ell_i \geq 2\frac{L}{t} \left|\left\{a : \ell_a > 2\frac{L}{t}\right\}\right| \ ,$$

hence $\frac{t}{2} \geq \left|\left\{a : \ell_a > 2\frac{L}{t}\right\}\right|$. □

We now design a scheduling algorithm in the non-preemptive model of tasks. Its work complexity is $\mathcal{O}(L+\alpha p\sqrt{t}+\alpha p\min\{f,t\})$ against the adaptive Strongly-Adaptive f-Bounded adversary.

Algorithm 15: DefTri; pseudo-code for processor v

```
initialize STATIONS to a sorted list of all $p$ names of processors;
initialize TASKS to a sorted list of all $t$ names of tasks;
initialize SUBTASKS to a sorted list of all subtasks;
initialize variable $d$ representing the length of an epoch;
initialize variable $\phi$ representing the length of a phase;
initialize $i := 0$;
repeat
    $d := p/2^i$;
    if $|TASKS| \geq d(d+1)/2$ then
        $\phi :=$ |SUBTASKS|/|TASKS| // set $\phi$ to current average length of a task
        execute TaPeBB($v, d,$ TASKS, STATIONS, $\phi$);
        update TASKS, STATIONS, SUBTASKS according to TaPeBB output;
    end
    else
        $i := i + 1$;
    end
until $|TASKS| = 0$;
```

At first glance, algorithm Deforming-Triangle MAC Scheduling (DefTri for short) is similar to ScaTri introduced in the previous section and on the top of its design there is the TaPeBB procedure for performing tasks. Roughly speaking, the algorithm, repetitively, tries to choose tasks that could be packed into a specific triangle (parameters of which are controlled by the algorithm) and feed them to TaPeBB. Furthermore, once again the main goal of the algorithm is to avoid redundant or idle work. Hence, the main feature is to examine whether there is an appropriate number of tasks to fill in an epoch of TaPeBB. However, as we are dealing with the non-preemptive model, tasks cannot be done in small pieces (subtasks).

Consequently, apart from checking the number of tasks (and possibly reducing the size of an epoch) the algorithm also changes the phase parameter, i.e., the duration between consecutive broadcasts — this is somehow convenient in order to know how the input tasks should be provided for TaPeBB. It settles a “framework” (epoch) with “pumped rounds” that should be filled in appropriately. Figure 4.4 illustrates the idea of DefTri.

DefTri has the phase parameter ϕ set to the average length of the current set of tasks.

According to Fact 9, at least half of the tasks do not exceed twice the average length. In what follows, there are p processors in such TaPeBB execution what is consistent with p slots of increasing lengths for performing tasks. Precisely, processor i is able to perform tasks which total length is at most $i\phi$ before its broadcast.

Similarly as in Section 4.1, we described the process of assigning tasks to processors from the algorithm perspective, i.e., we illustrated that the system needs to provide input to TaPeBB. Nevertheless, for the sake of clarity we assume that TaPeBB collects the appropriate input by itself according to the following rules.

We have a sorted list of all tasks and schedule them one by one, in an increasing order. The first, shortest task is scheduled for the processor that will broadcast first in TaPeBB. It has a slot of length ϕ. Then, the next task is considered and if it fits to the remainder of the first slot it is scheduled there and another task is considered. Otherwise, it is scheduled to the second processor with a slot of length 2ϕ. We continue according to the described rule until possible. The following fact summarizes, that we are able to feed TaPeBB with a significant number of tasks.

Fact 10. *Let t be the number of remaining tasks, which lengths sum up to L. Let $t > p(p+1)/2$, where p is the number of operational processors and $\phi = L/t$. For such parameters, the ratio between the capability of* TaPeBB *for performing tasks and the sum of tasks that we are able to schedule is constant.*

Proof. Observe that the number of tasks of length greater than $2L/t$ is at most $t/2$, according to Fact 9. It follows that at least $t/2 > p(p+1)/4$ tasks have lengths at most $2L/t$, therefore we may use them to fill in slots from 4 to $p/2$, each almost fully (i.e., without an additive loss of at most $2L/t$). Hence, the total length of the tasks allocated to these slots is at least $\sum_{i=4}^{p/2}(i-2)L/t = (p/2-4)(p/2)L/t$. Since the capability of TaPeBB for performing tasks for such parameters is $p(p+1)L/2t$, we end the proof. □

Theorem 10. DefTri *performs $\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p \min\{f, t\})$ work against the Strongly-Adaptive f-Bounded adversary.*

Proof. We analyze the algorithm similarly as in the proof of Theorem 9, dividing work into four categories: productive, wasted, active idle and passive idle.

- Case 1: iterations in which $|\texttt{TASKS}| \geq d(d+1)/2$.

For this case we have a great number of tasks to perform and the only subtlety is to take care of the appropriate phase length, that is adjusted before each epoch. Similarly as in Theorem 9, productive work resulting from successful task performance accrues L units of work and so does active idle work, generated by processors after successful transmissions. If a processor crashes, its work in the epoch is at most ϕp, which is equal to the maximal length of the epoch rescaled by ϕ, which is the phase parameter.

Because we have f possible crashes and ϕ might be α at most, then we may upper bound this kind of work as $\alpha p \min\{f, t\}$, taking into consideration that the number of tasks t may be less than the adversary's possibilities of crashes.

According to Fact 10 we may be sure about being able to schedule $t/2$ tasks in an epoch, which constitute a constant fraction of the capability for performing tasks in the epoch. Hence, if there are processors which had no tasks scheduled in an epoch, they generate passive idle work which also contributes to $\mathcal{O}(L)$.

- Case 2: iterations in which $|\texttt{TASKS}| < d(d+1)/2$.

When there are not enough tasks to provide the input to TaPeBB, then the size of a single epoch of TaPeBB is reduced in order to prevent excessive idle or redundant work. Obviously, the productive and active idle work for such case is $\mathcal{O}(L)$, while wasted work does not exceed $\alpha p \min\{f, t\}$.

Likewise in the proof of Theorem 9, we use a union bound over a logarithmic number of cases for the passive idle kind of work. Assume that the length of the epoch is set to $\frac{p}{2^i}$, for some i, and there are not enough subtasks to fully fill the input of TaPeBB, hence the length of the epoch is reduced with parameter $i+1$. The ratio between both of these cases is

$$\frac{\frac{\frac{p}{2^{i+1}}(\frac{p}{2^{i+1}}+1)}{2}}{\frac{\frac{p}{2^{i}}(\frac{p}{2^{i}}+1)}{2}} = \frac{1}{4}\,\frac{p+2^{i+1}}{p+2^{i}} \geq \frac{1}{4}\ ,$$

thus we conclude that the reduced epoch is at most 4 times smaller than the initial one. In what follows, the number of reduced length epochs sufficing to perform all the subtasks is constant. According to Fact 9, we are able to schedule at least half of the overall number of tasks, which is a constant fraction of the capability for performing tasks in the reduced epoch, as shown in Fact 10.

Taking these facts altogether and a similar reasoning as in Theorem 9 over the logarithmic number of cases, we have that

$$cp\sum_{i=0}^{\log p}\frac{\sqrt{t}}{2^i} = \mathcal{O}(p\sqrt{t})$$

phases of work are needed, where c is the constant number of epochs required for performing all the tasks. Once again: we have a sum over a logarithmic number of cases characterized by the epoch length, where at most p processors are operational, and they are working for c executions of TaPeBB. Since the length of a phase is at most α, we multiply the estimate above, having that passive idle work generates $\mathcal{O}(\alpha p\sqrt{t})$ units of work.

We conclude that DefTri has $\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p\min\{f, t\})$ work complexity. □

4.3 Towards randomized solutions

We have already shown that preemption is helpful when considering arbitrary lengths of tasks that need to be done by a distributed system. However, a natural question appears whether we may perform even better when randomization is available for the considered problem and its corresponding model. The answer seemed to be positive for the non-adaptive adversary, hence we investigate this issue by adapting algorithm ROBAL from Chapter 3 and analysing it for the configuration focusing on tasks with preemption, providing an improved algorithm, some intuitions about it and a detailed analysis.

4.3.1 Algorithm RanScaTri

The name of RanScaTri follows from ScaTri and that in fact the actual task performing remains the same. Furthermore, assigning tasks to TaPeBB in RanScaTri follows the same rules as in ScaTri. What distinguishes both algorithms is the use of randomization. On the other hand, RanScaTri is an adapted ROBAL algorithm from Section 3.1. Hence, in order to avoid duplicating the same results, we concentrate on main differences and argue that RanScaTri works for tasks with preemption against a Non-Adaptive f-Bounded adversary.

Let us recall, that a Non-Adaptive f-Bounded adversary has to select a subset of f

Algorithm 16: RanScaTri, pseudo-code for processor v

```
if $p \leq \sqrt{L}$ then
    execute ScaTri;
end
else
    initialize TASKS to a sorted list of all $t$ tasks;
    initialize STATIONS to a sorted list of all $p$ names of processors;
    initialize SUBTASKS to a sorted list of all subtasks;
    initialize variable $\phi := 1$ representing the length of a phase;
    if $\log p > e^{\frac{\sqrt{L}}{32}}$ then
        execute a $L$-phase epoch where every processor has all tasks assigned (without
          transmissions);
        execute Confirm-Work-RST;
    end
    else
        $i = 0$;
        repeat
            if $(\frac{p}{2^i} \leq \sqrt{L})$ then
                Execute ScaTri;
            end
            if Mix-And-Test-RST$(i, L, p)$ then
                repeat
                    Execute TaPeBB$(v, \sqrt{L}, \mathtt{TASKS}, \mathtt{STATIONS}, \phi)$;
                    update TASKS, STATIONS, SUBTASKS according to the output
                      information from TaPeBB;
                until more than $\frac{3}{4}\sqrt{L}$ broadcasts are unheard throughout the whole loop;
            end
            increment $i$ by 1;
        until $i = \lceil \log p \rceil$;
    end
end
```

Algorithm 17: Mix-And-Test-RST, pseudo-code for processor v

```
Input: $i, L, p$
coin := $\frac{p}{2^i}$;
for $\sqrt{L} \log p$ times do
    if $v$ has not been moved to the front of list STATIONS yet then
        toss a coin with the probability coin$^{-1}$ of heads to come up
    end
    if heads came up in the previous step then
        broadcast $v$ via the channel and attempt to receive a message
    end
    if some processor $w$ was heard then
        move processor $w$ to the front of list STATIONS;
        decrement coin by 1;
    end
end
if at least $\sqrt{L}$ broadcasts were heard then
    return true;
end
else
    return false;
end
```

Algorithm 18: CONFIRM-WORK-RST, pseudo-code for processor v

```
$i := 0$ ;
repeat
    coin := $\frac{p}{2^i}$ ;
    toss a coin with the probability $\texttt{coin}^{-1}$ of heads to come up;
    if heads came up in the previous step then
        broadcast $v$ via the channel and attempt to receive a message;
    end
    if some processor $w$ was heard then
        clear list TASKS;
        break;
    end
    else
        increment $i$ by 1 ;
        if $i = \lceil \log p \rceil + 1$ then
            $i := 0$;
        end
    end
until a broadcast was heard;
```

processors prone to crashes and, additionally, needs to declare in which rounds the crashes of processors will occur. If we compare this scenario with the Linearly-Ordered f-Bounded adversary, against which ROBAL is designed, who, apart from choosing the faulty subset, chooses a linear order of crashes, then we may conclude that a Non-Adaptive f-Bounded adversary is a Linearly-Ordered f-Bounded one with an additional constraint. Thus, we conclude that ROBAL is resistant to a Non-Adaptive f-Bounded adversary.

In ROBAL, we used the TWO-LISTS algorithm as a sub-procedure, either when the number of tasks was accordingly greater than the number of operational processors, or for the case with just single epochs executed, when the number of tasks was low. In the former case the work formula of TWO-LISTS linearised in t, while in the latter we used a union bound over a logarithmic number of cases, in order to calculate work.

In RANSCATRI we use TAPEBB executed for $\sqrt{L}$ phases in the core part of the algorithm. However, since tasks have arbitrary lengths, we do not know what is the distribution of subtasks L over the number of tasks t. In other words, it is possible that there are just few very long tasks and performing them would take more than a constant number of epochs. Nevertheless, as described in Section 2.3.3, if the number of tasks is less than the number of phases for which TaPeBB is executed, then it ends earlier. In what follows, when there are few long tasks, then the analysis of work accrued for such condition is consistent with the analysis of Case 2 in the proof of Theorem 9.

For completeness, we present the following fact proving that whenever $\frac{p}{2^i} \leq \sqrt{L}$ holds, ScaTri is executed, with $\mathcal{O}(L + p\alpha)$ work.

Fact 11. *Let p be the number of operational processors, and L be the number of remaining subtasks. Then for $p \leq \sqrt{L}$* ScaTri *work complexity is $\mathcal{O}(L + p\alpha)$.*

Proof. If $p \leq \sqrt{L}$, then the remaining number of crashes is $f < p$, hence $f < \sqrt{L}$. ScaTri has $\mathcal{O}(L + p\sqrt{L} + p \ \min\{f, L\} + p\alpha)$ work complexity. In what follows the complexity is $\mathcal{O}(L + \sqrt{L}\sqrt{L} + \sqrt{L} \ \min\{\sqrt{L}, L\} + p\alpha) = \mathcal{O}(L + p\alpha)$. □

Theorem 11. RanScaTri *performs $\mathcal{O}(L + p\sqrt{L}\log p + p\alpha)$ expected work against the Non-Adaptive f-Bounded adversary.*

Proof. In the algorithm we are constantly controlling whether condition $\frac{p}{2^i} > \sqrt{L}$ holds. If not, then we execute ScaTri which complexity is $\mathcal{O}(L + p\alpha)$ for such parameters.

If this condition does not hold initially then we check another one i.e., whether $\log p > e^{\frac{\sqrt{L}}{32}}$ holds. For such configuration we assign all the tasks to every processor. The work accrued during such a procedure is $\mathcal{O}(pL)$. However, when $\log p > e^{\frac{\sqrt{L}}{32}}$, then together with the fact that $e^x > x^2/2$ we have that $\log p > L$ and so the total complexity is $\mathcal{O}(p \log p)$.

Finally, successful processors, that performed all the tasks have to confirm this fact by executing Confirm-Work-RST. We demand that only one processor will transmit and if this happens the algorithm terminates. The expected value of a geometric random variable lets us assume that this confirmation will happen in expected number of $\mathcal{O}(\log p)$ rounds, generating $\mathcal{O}(p \log p)$ work.

When none of the conditions mentioned above hold, we proceed to the main part of the algorithm. The testing procedure by Mix-And-Test-RST for each of disjoint cases, where the number of operational processors $P \in (\frac{p}{2^i}, \frac{p}{2^{i-1}}]$ requires a certain amount of work that can be bounded by $\mathcal{O}(p\sqrt{L}\log p)$, as there are $\sqrt{L}\log p$ testing phases in each case and at most $\frac{p}{2^i}$ processors take part in a single testing phase for a certain case.

In the algorithm we run through disjoint cases where the number of operational processors $P \in (\frac{p}{2^i}, \frac{p}{2^{i-1}}]$. From Lemma 3 we know that when some of the leaders were crashed, then a proportional number of all the processors had to be crashed with high probability. When leaders are crashed but the number of operational processors still remains in the same interval, then the lowest number of subtasks will be confirmed if only the initial segment of processors transmit. As a result, when half of the leaders were crashed, then the system still confirms $\frac{L}{8} = \Omega(L)$ subtasks. This means that even if so many crashes occurred, $\mathcal{O}(1)$ epochs still suffice to do all the tasks. Summing work over all the cases may be estimated as $\mathcal{O}(p\sqrt{L})$ for the cases, where $\sqrt{L}$ is not less than the number of remaining tasks. Otherwise, we are dealing with a situation, when there are few long tasks, that might need to be performed in more than a constant number of TaPeBB executions - similarly as in the proof of Case 2 of Theorem 9.

Let us recall the fact from Section 2.3.3, that when $\sqrt{L} > |\texttt{TASKS}|$, then the duration of a TaPeBB epoch is $|\texttt{TASKS}|$. Tasks are always provided to TaPeBB in a sorted way, hence shorter ones are scheduled to processors, which are going to broadcast earlier. The length of the longest task is α, so it is always trimmed by one subtask, excluding crashes. In what follows, α rounds suffice to perform all tasks within cases where $\sqrt{L} > |\texttt{TASKS}|$, generating $\mathcal{O}(p\alpha)$ work at most. Clearly, failing work in such case, resulting from crashes, does not exceed $\mathcal{O}(p\alpha)$ as well.

By Lemma 5 we conclude that the expected work complexity is bounded by:

$$\left((\log p)^2 \;\max\{e^{-\frac{1}{8}\sqrt{L}}, e^{-c\,(\sqrt{L}\log p)}\}\right)\mathcal{O}(pL + p\sqrt{L}\log^2 p)$$

$$+\left(1-(\log p)^2 \;\max\{e^{-\frac{1}{8}\sqrt{L}}, e^{-c\,(\sqrt{L}\log p)}\}\right)\mathcal{O}(p\sqrt{L}\log p)) = \mathcal{O}(p\sqrt{L}\log p) \;\;,$$

where the first expression comes from the fact, that if we entered the main loop of the algorithm then we know that we are in a configuration where $\log p \leq e^{\frac{\sqrt{L}}{32}}$. Thus we have that

$$\frac{pL + p\sqrt{L}\log^2 p}{e^{\frac{\sqrt{L}}{8}}} \leq \frac{pL + pL\log^2 p}{e^{\frac{\sqrt{L}}{16}}e^{\frac{\sqrt{L}}{16}}} \leq \frac{p + p\log^2 p}{e^{\frac{\sqrt{L}}{16}}} \leq p + p\log p = \mathcal{O}(p\log p) \;\;.$$

The second expression follows from the fact that the algorithm was in a condition where $\sqrt{L} \geq p$, and because it also holds that $p \geq \log p$ then consequently $\sqrt{L} \geq \log p$.

Altogether, we have $\mathcal{O}(L + p\alpha)$ work, resulting from reaching condition $\frac{p}{2^i} > \sqrt{L}$ and $\mathcal{O}(p\log p)$ work resulting from reaching $\log p \leq e^{\frac{\sqrt{L}}{32}}$. Additionally, there is $\mathcal{O}(L+p\sqrt{L}\log p+ p\alpha)$ work, because of the Mix-And-Test-RST procedure and the corresponding effort of the leaders, so overall the work complexity of RanScaTri is $\mathcal{O}(L + p\sqrt{L}\log p + p\alpha)$, what ends the proof. □

4.4 Comparison of results for the two models

In this section we carry out a brief comparison of the formulas that we proved in previous sections, i.e., the upper bound for preemptive scheduling from Theorem 9 vs the lower bound in the model of tasks without preemption from Corollary 3. This comparison shows the range of dependencies between model parameters for which both models are separated, i.e., the upper bound in the model with preemption is asymptotically smaller than the lower bound in the model without preemption. We settle the scope on the, intuitively greater, bound for the model without preemption and examine how the ranges of parameters influence the magnitude of the formulas. Let us recall both formulas:

Preemptive, upper bound: $\mathcal{O}(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$;

Non-preemptive, lower bound: $\Omega(L + \frac{L}{t}p\sqrt{t} + p\min\{f, t\} + p\alpha)$.

If L is the factor that dominates the bound in the non-preemptive model, then by simple comparison to other factors we have that L also dominates the bound in the preemptive model. In what follows both formulas are asymptotically equal when the total number of tasks is appropriately large.

When $\frac{L}{t}p\sqrt{t}$ dominates the non-preemptive formula, then by a simple observation we have that $1 \le \sqrt{\frac{L}{t}}$, because the number of tasks is greater than the number of subtasks, and applying a square root does not affect the inequality. Multiplying both sides of the inequality by $p\sqrt{L}$ gives $p\sqrt{L} \le \frac{L}{t}p\sqrt{t}$. Thus, if $\frac{L}{t}p\sqrt{t}$ dominates the bound in the non-preemptive formula then the non-preemptive formula is asymptotically greater than the preemptive one.

On the other hand when L dominates the bound then they are asymptotically equal, as both formulas linearize for such magnitude. These results confirm that in the channel without collision detection the non-preemptive model is more demanding for most settings of parameters.

Note that the upper bound obtained for randomized algorithm RanScaTri against non-adaptive adversary almost matches the absolute lower bound $\Omega(L + p\sqrt{L} + p\alpha)$ in this model, given in Lemma 13, and therefore beats both compared formulas for deterministic solutions.

4.5 Conclusions

In this chapter, we addressed the problem of performing tasks of arbitrary lengths on a multiple-access channel without collision detection. In particular, we investigated two scenarios: with and without preemption.

The first one was the model with preemption, where tasks can be done partially, even by different processors. Tasks with preemption are likely to be seen as chains of subtasks, that need to be done in a certain order. The natural bottleneck for such problem is therefore length α of the longest task. We showed a lower bound for the considered problem that is $\Omega(L + p\sqrt{L} + p\min\{f, L\} + p\alpha)$ and designed an algorithm that matches the proved bound, hence settled the problem. Additionally we considered a distinction between oblivious and non-oblivious tasks and showed that such a distinction does not matter with respect to the adaptive adversary.

On the other hand, we answered the question of how to deal with tasks without preemption, that need to be done in one piece in order to confirm progress. Here, as well, we showed a lower bound for the problem and developed a solution, based on the one for preemptive tasks, yet this turned out not to match the lower bound that we showed. The formula for work is $\mathcal{O}(L + \alpha p\sqrt{t} + \alpha p\min\{f, t\})$, while we proved the lower bound for the problem to be $\Omega(L + \frac{L}{t}p\sqrt{t} + p\min\{f, t\} + p\alpha)$.

We also showed that randomization helps in case of the Non-Adaptive adversary, lowering the cost of preemptive scheduling to the absolute lower bound $\Omega(L + p\sqrt{L} + p\alpha)$ with a logarithmic overhead, while it does not help against a more severe adaptive adversary.

Considering open directions for the research considered in this chapter, it is natural to address the question of channels with collision detection. It may happen that the distinction between oblivious and non-oblivious tasks will matter when such a feature is available. Furthermore, it is worth considering whether randomization could help improving the results while considering the non-preemptive setting or against adversaries of intermediate power, as it took place in Chapter 3.

Finally, the primary goal of this work was to translate scheduling from classic models to the model of a shared channel, in which it had not been considered in depth. Therefore, a natural open direction is to extend the model with other features considered in the scheduling literature.

Chapter 5

Partially ordered tasks

In this chapter, we examine tasks that are in a partial order relation and the resulting task dependencies have to be preserved during the execution. Partially ordered sets of our particular interest consist of sets of independent chains of arbitrary lengths and some types of trees. Communication takes place via a multiple-access channel and we assume that there is a Strongly-Adaptive adversary, interfering with the system.

In Section 5.1, we show the lower bound and a matching solution for performing partially ordered sets of tasks forming chains, which completes Do-All with work $\mathcal{O}(t + p\sqrt{t} + p\min\{f,t\} + pH)$, where t is the total number of tasks, p is the number of operational processors and H represents the number of tasks in the longest chain.

Based on the solution for chains we also show a lower bound and an upper bound for performing tree shaped partial orders in Section 5.2. We prove the work performance for this problem to be $\mathcal{O}(t\log p + p\sqrt{t\log p} + p\min\{f,t\} + pH\log p)$.

algorithm	**work**	**lower bound**
ChainPerformer	$\mathcal{O}(t + p\sqrt{t} + p\min\{f,t\} + pH)$ Sec 5.1 Thm 13	$\Omega(t + p\sqrt{t} + p\min\{f,t\} + pH)$ Sec 5.1 Thm 12
TreePerformer	$\mathcal{O}(t\log p + p\sqrt{t\log p} + p\min\{f,t\} + pH\log p)$ Sec 5.2 Thm 14	$\Omega(t + p\sqrt{t} + p\min\{f,t\} + pH)$ Sec 5.2 Cor 4

Table 5.1: Summary of main results in Chapter 5. ChainPerformer for solving partially ordered sets of chains of tasks and TreePerformer for solving partially ordered sets of tasks forming trees are deterministic algorithms analyzed against the Strongly-Adaptive f-Bounded adversary on a channel without collision detection.

Finally, in Section 5.3 we discuss a heuristic approach for solving an arbitrary partially ordered set of tasks and also discuss the outcome of implementing an OR policy on a dependent structure of tasks, where we consider that an arbitrary task can be performed if at least one of its children was already performed. We summarize briefly in Section 5.4.

5.1 Sets of chains

In this section we examine one of the simplest cases of a partially ordered set of tasks i.e., how to perform tasks that are in a relation forming sets of chains. This will serve as a building block for more complex structures, which we analyze in the following sections of this chapter.

In contrast to further cases, considered later, performing chain-ordered tasks can be reduced to the result dedicated to performing arbitrary length preemptive tasks on a shared channel. Thus, in this case we take advantage of this reduction and utilize results from Chapter 4 for constructing both an effective algorithm and a tight lower bound.

5.1.1 From arbitrary length preemptive tasks to partially ordered sets of chains

Considerations in Chapter 4 include the problem of performing t preemptive tasks of arbitrary lengths, by p processors. If performing some of the tasks takes more than one computational time unit, then a natural question arises, whether such tasks have to be performed fully by a specific processor (non-preemptive model), or any intermediate progress is remembered and tasks may be reclaimed by some other processor (preemptive model).

More precisely, by preemption we define the possibility of performing tasks in several pieces. Consider task a of length l_a. If processor v is to perform task a and performs x units of this task, then the remaining part of $l_a - x$ units of task a may be done by some other processor w.

Because time is divided into rounds, we can assume that there is some minimal length of a task that may be processed in one step in the preemptive setting. In what follows all tasks can be presented as a multiplicity of the minimal length. Hence, tasks are assumed to be built with minimal length units, called *subtasks* and in order to perform a task all its subtasks have to be performed consecutively one by one. Such view was the basis for solving tasks of arbitrary lengths for the preemptive setting in Chapter 4.

Having explained preemptive tasks of arbitrary lengths, the correspondence to partially ordered sets of chains is simple. *Subtasks* from the preemptive model will now represent *tasks* and respectively *tasks* will be substituted by *chains*.

These adjustments are actually only syntactic changes, that allow us to improve the presentation. Consequently, all the results translate straightforwardly, so we can state the lower bound for our problem:

Theorem 12. *(Chapter 4, Theorem 8) The Adaptive f-Bounded adversary, for $0 \leq f < p$, can force any reliable, possibly randomized and centralized algorithm and a partially ordered set of chains of tasks to perform $\Omega(t + p\sqrt{t} + p\min\{f, t\} + pH)$ work, where H represents the length of the longest chain of tasks.*

5.1.2 ChainPerformer

Algorithm ScaTri from Chapter 4 for the problem of performing t preemptive tasks of arbitrary lengths by p processors can be applied to our problem of performing partially ordered sets of tasks forming chains. Its design is based on reducing idle work, according to the current number of available tasks.

```
Algorithm 19: ChainPerformer; pseudo-code for processor $v$
  Input: v, lists: TASKS, CHAINS, STATIONS
  Output: changes on list STATIONS and all tasks performed
  Algorithm ScaTri;
```

ChainPerformer maintains data on three lists. List `STATIONS` serves as a register of operational processors - this helps the algorithm to schedule broadcasts accordingly. List `CHAINS` contains information about all t tasks and their distribution over chains. We assume that processors have access to the whole partial order. List `TASKS` contains information about tasks that have not been performed yet.

The result of running ChainPerformer is that it performs all chains of tasks, while preserving their order and, additionally, it records which processors did crash. The following theorem states the work performance of ChainPerformer:

Theorem 13. *(Chapter 4, Theorem 9) ChainPerformer performs $\mathcal{O}(t+p\sqrt{t}+p\min\{f,t\}+pH)$ work against the Strongly-Adaptive f-Bounded adversary and partially ordered sets of chains of tasks, where H represents the longest chain.*

For a full description of ChainPerformer please refer to Chapter 4.

5.2 Trees

In this section we analyze another type of a partially ordered structure. Firstly, we examine the nature of *trees growing upwards*, making the root the least node that has to be done at the very beginning. Then, we demonstrate how similar methods can be used for trees growing downwards with numerous least elements and a single greatest task that has to be done in the end, located in the root.

For the upwards growing tree, observe that near the root, it is better to direct all the effort to perform all initial tasks, in order to open access to a wider range of tasks which can be performed in parallel, with limited redundant or idle work.

The idea of our solution is based on decomposing the tree into a number of layers and then performing them consecutively. Hence, tasks from the initial layer (containing the root) must be all done before processors begin working on the following layer. In general processors can start performing tasks from layer i, when all tasks from layer $i-1$ are confirmed.

Each layer consists of disconnected trees that are independent (i.e., two trees in the same layer have incomparable tasks) due to a careful construction and can be executed in parallel.

5.2.1 Lower bound

We begin with a statement about the lower bound for the problem on a partial order forming a tree growing upwards.

Lemma 17. *A reliable algorithm, possibly centralized and randomized, performs $\Omega(pH)$ work even in an execution in which no failures occur, on a partial order of tasks forming an upwards growing tree, where H is the height of the tree.*

Proof. A tree shaped partial order of height H contains a chain of length H. If p processors are operational for H rounds, what is consistent with performing the longest chain, then they generate pH work.

□

Combining Theorem 12 together with the Lemma above leads us to the lower bound for work in the assumed model.

Corollary 4. *A reliable algorithm on a tree-shaped partial order of tasks has to perform* $\Omega(t + p\sqrt{t} + p\min\{f, t\} + pH)$ *work against the Strongly-Adaptive adversary.*

5.2.2 TreePerformer: construction

We begin this section with a construction allowing to decompose an arbitrary tree into a set of disjoint components.

Tree decomposition. Let us consider the following construction. We label each node in the tree by a pair (x, y) of natural numbers, representing node's *colour* and node's *counter*, respectively. Each leaf is labelled $(1, 1)$, i.e., its colour equals 1 and its counter is set to 1 as well. All non-leaf nodes are labelled recursively according to the procedure described below. Let us consider node v. Let x be the **maximal** colour over v's children.

- If exactly one child of v has colour x and its counter is y, then v is labelled as $(x, y+1)$.
- If $l > 1$ children of v have colour x, let $(x, y_1), (x, y_2), \ldots, (x, y_l)$ denote their labels. We consider two cases dependently on the value $y = \sum_{i=1}^{l} y_i$.
  - If $y \geq 2t/p$, then v is labelled as $(x + 1, 1)$.
    (We reset the counter and start a group with a new colour).
  - Otherwise, if $y < 2t/p$, then v is labelled as $(x, y + 1)$.
    (We add v to the group of nodes marked with colour x).

This procedure partitions the tree into sub-trees labelled with different colours. Let us consider the properties of this partition.

Lemma 18. *Let $\mathcal{T}$ be a tree labelled according to the procedure described above with t nodes and let H be the length of the longest path in $\mathcal{T}$.*

1. *None of the nodes will have colour greater than* $\log p + 1$.
2. *Every connected monochromatic component (i.e., connected set of nodes, labelled with the same colour) has at most $2t/p + H$ nodes.*

Proof. First, note that the counter of v's label is the number of successors of v with the same colour as v. Let $N(x)$ be the minimum number of successors of a node in $\mathcal{T}$ labelled with colour x. One can easily see that for $x > 1$ inequality $N(x) \geq \frac{2t}{p} + 2N(x-1)$ holds. That is, if a node is labelled with x, it has at least $\frac{2t}{p}$ successors with colour $x - 1$. The colour in a node can be set to x only if it has at least two children with colour $x - 1$ such that their counters sum up to at least $\frac{2t}{p}$. Each of them has at least $N(x - 1)$ successors. Moreover, we can point two of them that are "minimal", i.e., have no successor with colour $x - 1$. We get $N(x) \geq \frac{2t}{p} + 2\left(\frac{2t}{p} + 2\left(\frac{2t}{p} + \ldots + 2\left(\frac{2t}{p} + 2N(1)\right)\right)\right)$.

Let us recall that t is the total number of nodes. Since leafs are coloured with $x = 1$ and have no successor, we have $N(1) = 0$. We obtain $N(x) \geq \frac{2t}{p}\left(1 + 2^1 + \ldots 2^{x-2}\right) = \frac{2t}{p}(2^{x-1} - 1)$. Clearly, for any node the number of its successors cannot exceed $t - 1$, thus we have $N(x) \leq t - 1$ and, as a consequence, $\frac{2t}{p}(2^{x-1} - 1) \leq t - 1$. Equivalently, $x \leq \log p + 1$, what finishes the proof of the first point of the lemma.

To prove the second point of the lemma, let us note that a connected subset of a tree is a tree as well. Let $\mathcal{T}'$ be the biggest monochromatic subset of tree $\mathcal{T}$ rooted as some node r with label (x_r, y_r). One can see that the counter y_r of r is the size of $\mathcal{T}'$.

Let us consider two cases. In the first case we assume that r has at least two children with colour x_r. Note that in this case y_r was computed as the sum of children's counters plus 1. However the result cannot exceed $\frac{2t}{p}$.

In the second case, again we have two sub-cases. If $\mathcal{T}'$ is a path, then its size cannot exceed H. Otherwise, r has a successor v with at least two children in $\mathcal{T}'$. Again, v's counter y_v cannot exceed $\frac{2t}{p}$ and hence $y_r \leq y_v + k$, where k is the length of the path between r and v, which cannot exceed H. We proved that in any case $y_r \leq \frac{2t}{p} + H$.

□

TreePerformer sub-procedures. Let us recall, that processors have entire knowledge about the partial order of tasks. In what follows, processors may perform local computations regarding the partial order and hence, follow the construction above.

Using the aforementioned construction one can perform the tree decomposition in the following way. Let the i-th *layer* be a set of all nodes with colour i. Clearly each layer consist of monochromatic disjoint trees of size at most $\frac{2t}{p} + H$ each, according to Lemma 18. We refer to this procedure as Decompose-Tree in the pseudo-code of Algorithm 20.

Each such tree in a given layer can be *translated* into a chain in the following way - having a tree we construct a chain of the same nodes, that preserves the order of the tree,

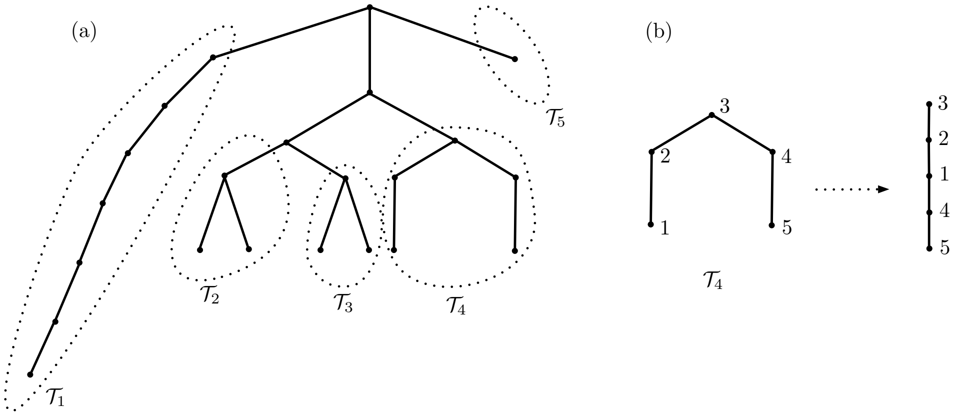

Figure 5.1: (a) Example decomposition of a tree. Layer 1 consists of 5 trees which can be performed in parallel. (b) Sub-tree into chain translation.

i.e., if $a \prec b$ in the tree then $a \prec b$ in the chain as well. Note, however, that it is possible that $a \prec b$ in the chain, while a and b are incomparable in the original tree. Clearly, such translation is not unique, except the case when the tree is a chain.

Consequently, each layer is mapped to a set of independent chains that can be performed by ChainPerformer. In order to preserve the clarity of the algorithm, in the pseudo-code we simply execute Translate-subtrees-into-chains to perform the above mentioned operation. To complete TreePerformer we execute ChainPerformer for each of the l consecutive layers.

Algorithm 20: TreePerformer; pseudo-code for processor v

```
initialize STATIONS to a sorted list of all $p$ names of processors;
initialize TASKS to a sorted list of all $t$ names of tasks;
initialize CHAINS to an empty list;
initialize LAYERS to an empty list;
$i := 0$;
LAYERS := Decompose-Tree;
repeat
    CHAINS = Translate-subtrees-into-chains(LAYERS$_i$);
    execute ChainPerformer($v$, CHAINS, TASKS, STATIONS);
    $i++$ ;
until $|TASKS| = 0$;
```

TreePerformer: analysis. Having all the tools, we can now proceed to the analysis of the algorithm.

Theorem 14. TreePerformer *performs* $\mathcal{O}(t \log p + p\sqrt{t \log p} + p \min\{f, t\} + pH \log p)$ *work for* p *processors and* t *tasks ordered in a tree of height* H *in presence of a Strongly-Adaptive* f*-Bounded adversary.*

Proof. TreePerformer decomposes a partial order using the technique described in Section 5.2.2. All tasks having a particular colour are grouped into a single layer. Hence, the total work depends on the number of layers and the work that is necessary to perform all the tasks within a single layer.

Let t_i be the number of tasks in the i-th layer and f_i be the number of processors crashed when the i-th layer is being performed. H_i denotes the largest sub-tree within layer i. Since the i-th layer consists of t_i tasks, which are translated into chains, so running ChainPerformer on this layer performs all the tasks with work W_i which does not exceed $O(t_i + p\sqrt{t_i} + p \min\{f_i, t_i\} + pH_i)$, according to Theorem 13.

On the other hand, we know that the longest chain in the i-th layer was constructed from a single sub-tree. Hence, by Lemma 18, we know that $H_i < 2t/p + H$.

The total work, calculated over all l layers does not exceed

$$\sum_{i=1}^{l} W_i \leq \sum_{i=1}^{l} \mathcal{O}(t_i + p\sum_{i=1}^{l}\sqrt{t_i} + p\sum_{i=1}^{l}\min\{f_i, t_i\} + p\sum_{i=1}^{l}(2t/p + H))$$

$$\leq \mathcal{O}(t + p\sum_{i=1}^{l}\sqrt{t_i} + p\min\{f, t\} + l \cdot t + lpH).$$

To estimate this further, we need the following fact, that can be easily proved using the Cauchy-Schwarz inequality:

Fact 12. *Let* $n_1 + n_2 + \ldots + n_l = n$ *and* $n_i > 0$ *for all* i*. Then* $\sum_{i=1}^{l} \sqrt{n_i} \leq \sqrt{l \cdot n}$.

Applying this fact to the estimate above gives us that

$$\mathcal{O}(t + p\sum_{i=1}^{l}\sqrt{t_i} + p\min\{f, t\} + l \cdot t + lpH) = \mathcal{O}(p\sqrt{lt} + p\min\{f, t\} + l \cdot t + lpH).$$

According to the first point of Lemma 18, we know that $l \leq \log p + 1$, so we conclude that the overall work of TreePerformer is $\mathcal{O}(t \log p + p\sqrt{t \log p} + p \min\{f, t\} + pH \log p)$, what ends the proof.

□

Towards generalized trees. TreePerformer answers the question of how to treat tree-shaped partial orders growing upwards. One may ask how to treat the other one: a tree growing from the top to the bottom.

The tree decomposition construction in 5.2.2 is independent of the direction in which the tree grows, so TreePerformer can be straightforwardly applied to solve a downwards growing tree with the root being the greatest node.

A generalized tree is a combination of a tree growing upwards and a tree growing downwards. Since we know how to solve both of them, then we conclude that a generalized tree can be also solved by TreePerformer with the same, asymptotically, work complexity. The only difference is that layers of the bottom component (downwards growing tree) should be performed in reverse order.

5.3 Extensions

A natural question is to extend the results from previous sections to an arbitrary order of tasks. In what follows, we present and discuss an approach dedicated to a different way of thinking about an arbitrary partial order. We complement these results with a study of a weaker restriction on performing tasks – a task could be performed if *some* of its direct predecessors has already been performed.

5.3.1 Layered representation of an arbitrary order

Algorithm 21: ArbitraryPerformer; pseudo-code for processor v

```
initialize STATIONS to a sorted list of all $m$ names of processors;
initialize LAYERS to an empty list;
LAYERS := Separate-Layers;
forall $l \in$ LAYERS do
    execute ChainPerformer($v, |l|, l,$ STATIONS);
end
```

Let us consider an arbitrary partially ordered set R from its Hasse diagram perspective. We can create a one-to-one projection of this set to its Hasse diagram, by compressing the diagram to a grid as follows: we begin with inserting minimal elements to the initial row, and then place elements inductively in the next row, which are (i) immediate successors of elements in the recently filled row, and (ii) which are not successors of other non-inserted nodes, together with edges corresponding to dependencies between elements in subsequent rows, cf., Figure 5.2. We call the rows of the grid *layers*. Relations are preserved in such a way that if there is a relationship between some two elements i.e., one precedes the other (in the sense of the specific relation), then they belong to different layers. Elements located in the same layer are incomparable. We represent the separation of layers in the pseudocode of Algorithm ArbitraryPerformer as procedure Separate-Layers, which generates a partition of tasks into a list of independent sets of tasks.

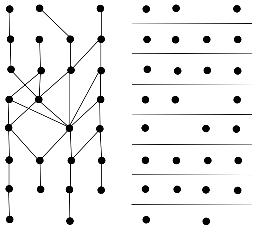

Figure 5.2: Correspondence between the Hasse diagram and its layered representation.

Having such a representation of a partially ordered set we may efficiently deal with a class of partial orders that are very dense i.e., there is a large number of relations between consecutive layers. In particular, there may be even complete bipartite graphs between any two consecutive layers, thus layer l_{i+1} cannot be performed unless all the tasks from layer l_i have been performed.

Theorem 15. ArbitraryPerformer *performs* $\mathcal{O}(t + p\sqrt{tH} + p\min\{f,t\})$ *work for* p *processors and* t *arbitrarily ordered tasks in presence of a Strongly-Adaptive adversary.*

Proof. Since all elements within a single layer are independent, then we may simply execute ChainPerformer separately for each layer. Independent elements are equivalent to chains of length 1, hence the complexity of ChainPerformer for a set of tasks l_i is $\mathcal{O}(t_i + p\sqrt{t_i} + p\min\{f_i, t_i\})$, where t_i is the number of elements in l_i and f_i is the number of crashes that occur while performing tasks in layer l_i. Thus, we can perform all tasks in the partial order with work $\mathcal{O}(\sum_{i=1}^{H}(t_i + p\sqrt{t_i} + p\min\{f_i, t_i\})) \leq \mathcal{O}(t + p\sqrt{tH} + p\min\{f, t\})$, where H denotes the number of layers in the partial order. The last inequality follows from

Fact 12 and that the total number of tasks is t while the upper bound on the number of crashes is f.

□

5.3.2 The OR policy of performing tasks

Partial order relations on sets of tasks that we considered throughout this paper are characterized by the necessity of performing all predecessors of a certain task before the particular task can be done. One could consider a relaxation of this policy to require only some of the immediate predecessors, or even only one of them, to be performed in order to begin executing the actual task. This policy corresponds to OR trees. (Note that in many popular system architectures, e.g., blockchains, quorum or network-coding based algorithms, obtaining only a fraction of prerequisites could be sufficient to trigger the next task.)

In an OR tree, where we say that there is an *OR policy*, performing just one of the immediate descendants of a particular element gives access to performing that element. In other words, if there is a task a which has many children $w_1, \ldots, w_j$ (immediately preceding tasks), then performing just one of them, say w_i, is sufficient to unlock access to performing a, what follows the rules of disjunction satisfiability. If we now consider again the layered representation of an arbitrary partial order with an OR policy available, then we can show that for some sets of tasks this helps to significantly increase the performance of the layered representation heuristic, comparing to the all-children-performed requirement considered earlier in this paper (by analogy, we refer to it in this part as the *AND policy*) – especially for partially ordered sets that have a large number of connections between consecutive layers.

Indeed, let us assume that layer l_j contains tasks $a_i^{l_j}$, where i represents the id of the task. Consider the following approach. When processor v is about to perform task $a_i^{l_1}$, then either it is crashed by the adversary or it performs this task. When $a_i^{l_1}$ is done, v may choose one of its successors from layer l_2 and the situation repeats until the whole path of tasks through all layers is performed. Having explained the details above, each station may be responsible for its own path of consecutive tasks, what actually means that this is consistent with the scenario of partially ordered sets of chains. Consequently, we already showed an efficient algorithm for solving chains with work $\mathcal{O}(t + p\sqrt{t} + p\min\{f,t\})$ and therefore, the OR policy results in a significant improvement in comparison to the general heuristic formula $\mathcal{O}(t + p\sqrt{tH} + p\min\{f,t\})$ introduced earlier in Section 5.3.1 for the classical AND policy.

5.4 Conclusions

In this chapter, we addressed the question of performing a set of t partially ordered jobs on a distributed system of p processors, communicating on a shared channel with adversarial distractions. Some results turned out to be closely correlated with the related problem of jobs having different lengths. We showed how to solve the problem on partial orders forming sets of chains. This solution occurred to match the lower bound and both the algorithm and the lower bound were a transformation of results for preemptive jobs with different lengths.

On the other hand, we showed how to deal with slightly more complicated structures i.e., trees. Our solution is based on a tree decomposition method which allowed to arrange the tree into a set of layers, which can be performed iteratively. The solution turned out to be logarithmically far from the proved lower bound. We also introduced a general heuristic for an arbitrary partial order and commented on the OR policy for sets of dependent jobs. Nevertheless, it is still interesting whether one can show a better solution for arbitrary partially ordered set of jobs given as the input.

Adversarial scenarios seemed to be a background feature of the considered model, hence another interesting open direction for future work consists of investigating whether there are some classes of adversaries that are able to make an explicit use of the impediment in the form of job dependencies and impose greater work to be done, even for some simple classes of partial orders.

Chapter 6

Consensus against Constrained adversaries

In this chapter, we consider the problem of reaching agreement in a distributed message-passing system prone to crash failures. Crashes are generated by Constrained adversaries - a Weakly-Adaptive$_\mathrm{C}$ adversary, who has to fix, in advance, the set of f crash-prone processes, and a k-Chain-Ordered$_\mathrm{C}$ adversary, who orders all the processes into k disjoint chains and has to follow this order when crashing them. Apart from these constraints, both of them may crash processes in an adaptive way at any time. While commonly used Strongly-Adaptive$_\mathrm{C}$ adversaries model attacks and Non-Adaptive$_\mathrm{C}$ ones - pre-defined faults, Constrained adversaries model more realistic scenarios when there are fault-prone dependent processes, e.g., in hierarchical or dependable software/hardware systems. In this view, our approach helps to understand fault-tolerant Consensus better in more realistic executions.

We propose time-efficient Consensus algorithms against such adversaries. We complement our algorithmic results with (almost) tight lower bounds, and extend the one for Weakly-Adaptive$_\mathrm{C}$ adversaries to hold also for (syntactically) weaker Non-Adaptive$_\mathrm{C}$ adversaries. Together with the Consensus algorithm against Weakly-Adaptive$_\mathrm{C}$ adversaries (which automatically translates to the Non-Adaptive$_\mathrm{C}$ adversaries), these results extend the state-of-the-art of the popular class of Non-Adaptive$_\mathrm{C}$ adversaries, in particular, the result of Chor, Meritt and Shmoys [34], and prove a separation gap between Constrained adversaries (including Non-Adaptive$_\mathrm{C}$ ones) and Strongly-Adaptive$_\mathrm{C}$ adversaries, analyzed by Bar-Joseph and Ben-Or [14] and others.

Table 6.1 presents time complexities of solutions for the Consensus problem against

		Strongly-Adaptive$_{\mathrm{C}}$	**Weakly-Adaptive$_{\mathrm{C}}$** and **Non-Adaptive$_{\mathrm{C}}$**	k**-Chain-Ordered$_{\mathrm{C}}$**
randomized	upper bound	$\mathcal{O}\left(\sqrt{\frac{n}{\log n}}\right)$ [14]	$\mathcal{O}\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ *	$\mathcal{O}\left(\sqrt{\frac{k}{\log k}}\log(n/k)\right)$ *
	lower bound	$\Omega\left(\sqrt{\frac{n}{\log n}}\right)$ [14]	$\Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ *	$\Omega\left(\sqrt{\frac{k}{\log k}}\right)$ *
deterministic	upper bound	$f+1$ [47]		
	lower bound	$f+1$ [43]		

Table 6.1: Time complexity of solutions for the Consensus problem against different adversaries. Formulae with * are presented in this chapter.

different adversaries. Results for the Strongly-Adaptive$_{\mathrm{C}}$ adversary and for deterministic algorithms are known (cf., Section 1.2), while the other ones are delivered in this chapter.

We design and analyze a randomized algorithm, which reaches Consensus in expected $\mathcal{O}\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ rounds when a sub-procedure of the algorithm is instantiated with the Ben-Or and Bar-Joseph's protocol from [14], using $\mathcal{O}\left(n^2 + \sqrt{\frac{n^5}{(n-f)^5\log(n/(n-f))}}\right)$ point-to-point messages, in expectation, against any Weakly-Adaptive$_{\mathrm{C}}$ adversary that may crash up to $f < n$ processes. This result is time optimal due to the proved lower bound $\Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ on the expected number of rounds.

The lower bound could be also generalized to hold against the (syntactically) weaker Non-Adaptive$_{\mathrm{C}}$ adversaries, therefore all results concerning Weakly-Adaptive$_{\mathrm{C}}$ adversaries delivered in this paper hold for Non-Adaptive$_{\mathrm{C}}$ adversaries as well. This extends the state-of-the-art of the study of Non-Adaptive$_{\mathrm{C}}$ adversaries done in high volume of previous work, c.f., [34, 30, 53], specifically, an $O(1)$ expected time algorithm of Chor et al. [34] only for a constant (smaller than 1) fraction of failures. Our lower bound is the first non-constant formula depending on the number of crashes proved for this adversary. In view of the lower bound $\Omega\left(\frac{f}{\sqrt{n\log n}}\right)$ [14] on the expected number of rounds of any Consensus algorithm against a Strongly-Adaptive$_{\mathrm{C}}$ adversary crashing at most f processes, our result shows a separation between the two important classes of adversaries – Non-Adaptive$_{\mathrm{C}}$ and Strongly-Adaptive$_{\mathrm{C}}$ – for the Consensus problem, which is one of the most fundamental problems in distributed computing.

Furthermore, we show an extension of the above mentioned algorithm accomplishing Consensus in slightly (polylogarithmically) longer number $\mathcal{O}\left(\sqrt{\frac{n\log^5 n}{(n-f)}}\right)$ of expected rounds but using a substantially smaller number of $\mathcal{O}\left(\left(\frac{n}{n-f}\right)^{3/2}\log^{7/2} n + n\log^4 n\right)$ point-to-point messages in expectation. Time and message complexities of the latter algorithm are only polylogarithmically far from the lower bounds for almost all values of f. More specifically, only for $f = n - \Omega(\sqrt{n})$ the message complexity is away from the lower bound by some small polynomial.

6.1 Weakly-Adaptive$_\mathrm{C}$ adversary

In this section, we consider the fundamental result of this chapter i.e., WACons, which consists of two main components - a leader election procedure, and a reliable Consensus protocol. We combine them together in an appropriate way, in order to reach Consensus against a Weakly-Adaptive$_\mathrm{C}$ adversary.

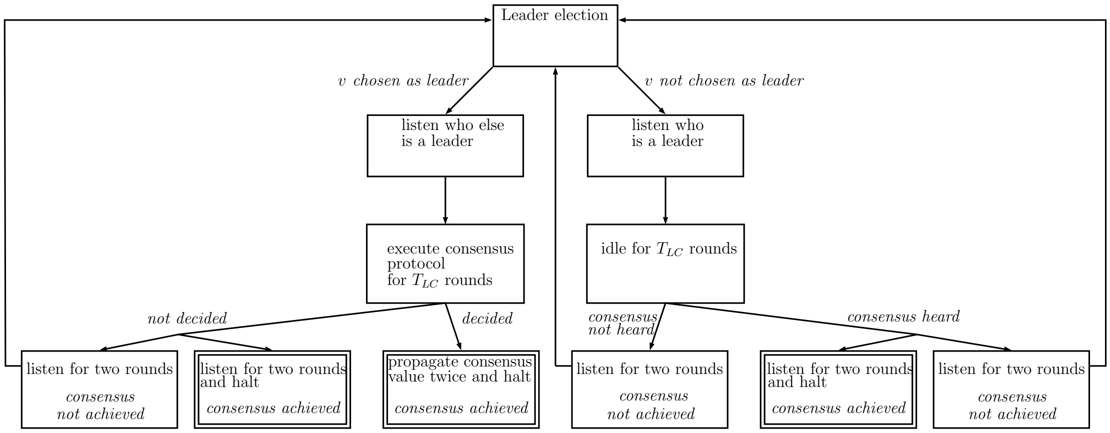

Figure 6.1: WACons flow diagram for process v.

WACons algorithm has an iterative character and begins with a leader election procedure in which we expect to elect $\mathcal{O}(\frac{n}{n-f})$ leaders simultaneously. Leaders run the Leader-Consensus procedure in which they reach Consensus within their own group with a constant probability. If they do so, this fact is communicated to all processes via Propagate-Msg, so that all processes that were not in the leaders group, know about Conditional-Consensus being reached and set their consensus values accordingly. Communicating this fact, implies reaching Consensus by the whole system. There are several subtle points in this intuitive description to be clarified, what we do next.

Let us follow Algorithm 22 from the perspective of some process v (cf., Figure 6.1, for

Algorithm 22: WACons, pseudo-code for process v

```
initialize list LEADERS to an empty list;
$decided := false$;
$value := x_v$;
repeat
    LEADERS := Elect-Leader-Cons;
    if LEADERS contains $v$ then
        $(decided, value)$ := Leader-Consensus($T_{LC}$(|LEADERS|), $value$) ;
        if $decided$ then
            execute Propagate-Msg($value$) twice;
        end
        else
            if heard the same consensus value $CV_w$ twice from some process $w$ then
                $value := CV_w$;
                $decided = true$;
            end
            if heard consensus value $CV_w$ once from some process $w$ then
                $value := CV_w$;
            end
        end
    end
    else
        idle for $T_{LC}$ rounds;
        if heard the same consensus value $CV_w$ twice from some process $w$ then
            $value := CV_w$;
            $decided = true$;
        end
        if heard consensus value $CV_w$ once from some process $w$ then
            $value := CV_w$;
        end
    end
    clear list LEADERS;
until $decided$;
```

Algorithm 23: Elect-Leader-Cons, pseudo-code for process v

```
coin := $\frac{1}{n-f}$;
initialize list LEADERS to an empty list;
toss a coin with the probability coin of heads to come up;
if heads came up in the previous step then
    Propagate-Msg(“$v$”) to all other processes;
    add $v$ to list LEADERS;
end
fill in list LEADERS with elected leaders’ identifiers from received messages;
return LEADERS;
```

convenience). At the beginning of the protocol every process takes part in Elect-Leader-Cons procedure and process v tosses a coin with probability of success equal $\frac{1}{n-f}$. It is either chosen to the group of leaders or not. If it is successful, then it communicates this fact to all processes.

Process v takes part in Leader-Consensus together with other leaders, in order to reach a Conditional-Consensus, which happens with a constant probability. Hence, if Leader-Consensus is successful and the consensus value is fixed, v tries to convince other processes to this value twice. This is because if some process $w \neq v$ receives the consensus value (obtained from Leader-Consensus) in the latter round, then it may be sure that other processes received this value from v as well in the former round (so in fact every process has the same consensus value fixed from that point). Process v could not propagate its value for the second time if it was not successful in propagating this value to every other process for the first time – if just one process did not receive the value, this would indicate a crash of v.

However, if Leader-Consensus is unsuccessful in agreeing on a common value, the procedure is terminated after a certain number of rounds, which is fixed as an input value for Leader-Consensus. Even though Conditional-Consensus was not reached, it might happen that some of the processes, including v, terminate the procedure with a decided value. In what follows, these processes propagate the decided value to all other processes, similarly as in the successful case.

On the other hand, if process v was not chosen to be a leader, then it listens to the channel for an appropriate amount of time and afterwards tries to learn the consensus value twice. If it is unable to hear the value twice, then it is consistent with being idle for two rounds. If Consensus is not reached, then the protocol starts again with electing another group of leaders. Nevertheless, if process v hears a consensus value once, it holds and assigns it as a candidate consensus value. This guarantees the continuity of the protocol and its validity.

The idea standing behind WACons is built on the fact that if just one fault-resistant process is elected to the group of leaders then the adversary is unable to crash it in the course of an execution, and hence Consensus is achieved after a certain expected number of rounds.

6.1.1 Analysis of WACons

We begin with a lemma stating that Elect-Leader-Cons is likely to elect $\mathcal{O}(\frac{n}{n-f})$ leaders, with a certain probability.

Lemma 19. *Let L denote the number of leaders elected by* Elect-Leader-Cons. *$L < 6\frac{n}{n-f}$ at least with probability $\frac{9}{10}$.*

Proof. Let X be a random variable such that $X = X_1 + \cdots + X_n$, where $X_1, \cdots, X_n$ are Poisson trials and

$$X_i = \begin{cases} 1 & \text{if station i chosen a leader,} \\ 0 & \text{otherwise.} \end{cases}$$

We know that

$$\mu = \mathbb{E}X = \mathbb{E}X_1 + \cdots + \mathbb{E}X_n = \frac{n}{n-f}.$$

To estimate the probability that the number of leaders lies within the desired interval we use the following Chernoff's inequality: $\mathbb{P}[X \geq R] \leq 2^{-R}$, where $R \geq 6\mu$.
Let $R = 6\frac{n}{n-f}$. Then

$$\mathbb{P}\left[X \geq 6\frac{n}{n-f}\right] \leq 2^{-6\frac{n}{n-f}} \leq \frac{1}{2^6} \leq \frac{1}{10}.$$

In what follows, at least with probability $\frac{9}{10}$ the number of leaders is less than $6\frac{n}{n-f}$.

□

Lemma 20. *Procedure* Elect-Leader-Cons *elects at least one fault-resistant leader with probability at least $\frac{6}{10}$.*

Proof. There are n processes out of which f are prone to crashes. Hence, the system contains $n-f$ fault-resistant processes. The probability that a fault-resistant one will respond in the election procedure is $\frac{1}{n-f}$. We estimate the probability of electing at least one fault-resistant leader by a complementary event that none of the fault-resistant processes responded during the election procedure:

$$\left(1-\left(1-\frac{1}{n-f}\right)^{n-f}\right) \geq (1-e^{-1}) \geq \frac{6}{10}.$$

□

Theorem 16. *The following statements are true about* WACons *against the Weakly-Adaptive$_C$ adversary:*
a) WACons *reaches Consensus in the expected number of rounds equal to* $\mathcal{O}\left(T_{LC}\left(\frac{n}{n-f}\right)\right)$*, satisfying termination, agreement and validity.*
b) *Expected work per process of* WACons *is* $\frac{1}{n-f}\ \mathcal{O}\left(T_{LC}\left(\frac{n}{n-f}\right)\right)$.
c) *For some fixed* $\epsilon > 0$*,* WACons *reaches Consensus in* $\mathcal{O}\left(\log\left(\frac{1}{\epsilon}\right)\ T_{LC}\left(\frac{n}{n-f}\right)\right)$ *rounds and with work per process* $\frac{1}{n-f}\ \mathcal{O}\left(\log\left(\frac{1}{\epsilon}\right)\ T_{LC}\left(\frac{n}{n-f}\right)\right)$ *with probability* $1-\epsilon$.

Proof. Let us describe three events and their corresponding probabilities:

- H - (Lemma 19) Elect-Leader-Cons chooses less than $6\frac{n}{n-f}$ leaders, with probability greater than $\frac{9}{10}$.
- U - (Lemma 20) Elect-Leader-Cons chooses at least one fault-resistant leader, with probability greater than $\frac{6}{10}$.
- J - (cf., Section 2.3.4) Leader-Consensus reaches agreement within its expected time $T_{LC}(\frac{6n}{n-f})$, with probability $\frac{9}{10}$.

Let us define the conjunction of these events as the *success of the algorithm* and denote it as Z. Since for any events A and B we have that $\mathbb{P}(A \cap B) \geq \mathbb{P}(A) + \mathbb{P}(B) - 1$, it is easy to observe that Z happens with probability at least $\frac{2}{5}$.
a) If Z holds then WACons reaches Consensus in time $\mathcal{O}(T_{LC}(\frac{6n}{n-f}))$. In what follows, since Z takes place with probability $\frac{2}{5}$, then $\frac{5}{2}$ is the expected number of iterations of WACons until reaching Consensus. Consequently WACons reaches Consensus in the expected number of rounds equal to $\mathcal{O}(T_{LC}(\frac{6n}{n-f}))$.

Because the termination condition is satisfied by LEADER-CONSENSUS and the expected number of iterations of WACONS is a constant, WACONS also meets the termination condition, provided that event J holds. However, if it does not hold, then LEADER-CONSENSUS is terminated after a fixed number of rounds, so termination is satisfied as well.

The validity condition is also satisfied, by the validity of the LEADER-CONSENSUS protocol and the fact that WACONS does not change values held by processes.

In order to prove the agreement condition let us assume that event J holds and suppose to the contrary that in some execution $\mathcal{A}$ there is process v that reached Consensus with value 1 and process w that reached Consensus with value 0. Process v did set value 1 as a result of Conditional-Consensus reached by the LEADER-CONSENSUS procedure. So did process w with value 0. This contradicts the agreement condition of LEADER-CONSENSUS. Let us now assume, that event J does not hold and LEADER-CONSENSUS terminates deterministically. If some processes finished with a decided value, then by the conditional agreement condition of LEADER-CONSENSUS (cf., Section 2.3.4), we know that they finished with the same value. Now, if they successfully propagate their values twice, then all operational processes end with the same value and, hence, reach agreement.

b) In each iteration of WACONS a process either becomes a leader or stays idle, waiting to hear and adopt the consensus value decided by leaders. This leads to an observation that the actual work is done by processes taking part in LEADER-CONSENSUS. We know that $\frac{5}{2}$ is the expected number of iterations of WACONS and each process becomes a leader in each iteration with probability $\frac{1}{n-f}$. The expected number of rounds in each iteration is $\mathcal{O}(T_{LC}(\frac{6n}{n-f}))$, hence by applying Wald's equation ([85], Theorem 12.3) we have that $\frac{1}{n-f}\ \mathcal{O}(T_{LC}(\frac{6n}{n-f}))$ is the expected work per process.

c) We know that $\mathbb{P}(Z) \geq \frac{2}{5}$. Let $\epsilon > 0$ be fixed and $X \sim Geo\left(\frac{2}{5}\right)$. We have that $\mathbb{P}(X \geq i) = \left(1-\frac{2}{5}\right)^{i-1}$.

Taking $i = \frac{5}{2}\log\left(\frac{1}{\epsilon}\right)+1$, we have that $\mathbb{P}\left(X \geq \frac{5}{2}\log\left(\frac{1}{\epsilon}\right)+1\right) \leq e^{-\log\left(\frac{1}{\epsilon}\right)} = \epsilon$. Thus, $\mathbb{P}\left(X < \frac{5}{2}\log\left(\frac{1}{\epsilon}\right)+1\right) > 1-\epsilon$.

Applying this to results from points a) and b) of the proof gives the desired result. □

Corollary 5. *Instantiating* LEADER-CONSENSUS *with* SYNRAN *from [14] results in:*

a) $\mathcal{O}\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ *expected rounds to reach Consensus by* WACONS.

b) *Expected work per process equals* $\mathcal{O}\left(\sqrt{\frac{n}{(n-f)^3\log(n/(n-f))}}\right)$, *c)* $\mathcal{O}\left(\log\left(\frac{1}{\epsilon}\right)\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ *rounds to reach Consensus by* WACons *with probability* $1-\epsilon$*, for* $\epsilon > 0$*.*

WACons is designed in a way that most of the processors wait until the consensus value is decided and propagated through the communication medium. Expected work per process value leads to an observation that, considering more practical scenarios, most of the processors can take care about different tasks, while only a subset is responsible for actually reaching Consensus.

6.1.2 Improving message complexity

The expected number of point-to-point messages sent by processes during the execution of WACons could be as large as the expected time $\mathcal{O}\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ multiplied by $\mathcal{O}(n^2)$ messages sent each round by the communication procedure Propagate-Msg, which in the rough calculation could yield $\mathcal{O}\left(n^2\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ of total point-to-point messages sent. Estimating more carefully, each round of Propagate-Msg outside the Leader-Consensus execution costs $\mathcal{O}(n^2)$ point-to-point messages, and each message exchange inside the Leader-Consensus costs $\mathcal{O}\left(\frac{n^2}{(n-f)^2}\right)$ with constant probability, which gives a more precise upper bound $\mathcal{O}\left(n^2+\sqrt{\frac{n^5}{(n-f)^5\log(n/(n-f))}}\right)$ on the expected number of point-to-point messages.

In order to improve message complexity, we modify WACons in the following way – we call the resulting protocol MSGCons. Each communication round is implemented by using a gossip protocol from [29], which is a deterministic protocol guaranteeing successful exchange of messages between processes which are non-faulty at the end of this protocol in $O(\log^3 n)$ rounds using $O(m\log^4 n)$ point-to-point messages, where $m \leq n$ is the known upper bound on the number of participating processes. This gives $\mathcal{O}(n\log^4 n)$ point-to-point messages per execution of Propagate-Msg outside of Leader-Consensus procedure, and $\mathcal{O}\left(\frac{n\log^4 n}{n-f}\right)$ point-to-point messages, with constant probability, in each exchange during the execution of Leader-Consensus. Therefore, MSGCons accomplishes consensus in $\mathcal{O}\left(\sqrt{\frac{n\log^5 n}{(n-f)}}\right)$ expected rounds using $\mathcal{O}\left(\left(\frac{n}{n-f}\right)^{3/2}\log^{7/2} n + n\log^4 n\right)$ point-to-point messages in expectation. In particular, for a large range of crashes $f = n - \Omega(\sqrt{n})$, MSGCons is nearly optimal (i.e., with respect to a polylogarithm of n) in terms of both expected time and message complexity.

6.1.3 Lower bound

Theorem 17. *The expected number of rounds of any Consensus protocol executed against a Weakly-Adaptive*$_C$ *or a Non-Adaptive*$_C$ *adversary causing up to* f *crashes is* $\Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$.

Proof. Consider a Consensus algorithm. We partition n processes into $m = \frac{n}{n-f}$ groups of size $n-f$ and treat each group as a distributed entity, similarly as in the proof of the impossibility result for Consensus against at least third part of Byzantine processes, cf., [9].

We assume that entities are described by a tuple, where each position in the tuple corresponds to a certain processor and the elements of the tuple describe processors' states. Furthermore, we assume that communication between entities is by sending concatenated messages of all the processes gathered in a particular round.

Bar-Joseph and Ben-Or showed in [14] that the expected number of rounds of any Consensus protocol running against a Strongly-Adaptive$_C$ adversary causing up to $m/2$ crashes on m processes is $\Omega\left(\sqrt{m/\log m}\right)$. We call this adversary the Bar-Joseph and Ben-Or adversary, for reference. It automatically implies, by Markov inequality, that with probability at least $3/4$ an execution generated by the Bar-Joseph and Ben-Or adversary has length $\Omega\left(\sqrt{m/\log m}\right)$. We apply this to the above mentioned system of m groups. This leads us to a class of executions denoted as $\mathcal{A}_s$, generated by the Strongly-Adaptive adversary, which simulates all the executions of given algorithms until half of the entities remain operational. We consider a class $\mathcal{A}_s$ as an execution tree that we define as follows.

Let us assume, that the execution tree of an algorithm is a tree where each path from the root to the leaves represents the number of rounds a particular execution takes. What is more, nodes are labeled with random bits and the status of entities (which of them are operational) at a particular round of the execution. Note that possible system configurations at round i are represented as nodes at level i of the tree. This may lead to an enormously huge tree, with each path/node having its probability of occurrence; yet we assume that the adversary has sufficient computational power to build such a structure.

We call such an execution *long* if it has length $\Omega\left(\sqrt{m/\log m}\right)$. Recall, that the probability of an event that an execution against the Bar-Joseph and Ben-Or adversary is in class $\mathcal{A}_s$ and is long is at least $3/4$.

The tree is built along paths (corresponding to specific executions) until half of the entities remain operational, hence there is an entity (i.e., group) E that remained operational in at least half of these executions (or more precisely, in executions which occur with probability at least 1/2), again by using a pigeonhole type of arguments with respect to the remaining groups and executions. It follows that with probability at least $3/4 - 1/2 = 1/4$ a long execution in $\mathcal{A}_s$ contains group E.

Let us now consider a Weakly-Adaptive$_{\mathrm{C}}$ adversary that chooses E as the group of $n - f$ fault-resistant processes, before the execution of the algorithm. We consider a class of executions $\mathcal{A}_w$ generated by the Weakly-Adaptive$_{\mathrm{C}}$ adversary of Bar-Joseph and Ben-Or, by simulating the same executions (until possible) as the Strongly-Adaptive$_{\mathrm{C}}$ one considered above. Note that such adversary crashes whole groups. With probability at least 1/4 there is an execution containing E (therefore, the Weakly-Adaptive$_{\mathrm{C}}$ adversary does not stop it before the Strongly-Adaptive$_{\mathrm{C}}$ does) and of length $\Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$. This implies the expected length of an execution caused by the Weakly-Adaptive$_{\mathrm{C}}$ adversary not failing group E to be at least $1/4 \cdot \Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right) = \Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$.

In order to extend this bound into the model with a Non-Adaptive$_{\mathrm{C}}$ adversary, we recall that in the proof of the lower bound in [14], in each round a suitable set of crashes could be done with probability $1 - O(1/\sqrt{m})$. Therefore, by a union bound over $\Theta(\sqrt{m/\log m})$ rounds, there is a path in the tree used in the first part of this proof which leads to $\Theta(\sqrt{m/\log m})$ rounds without Consensus, with probability at least $1 - 1/\log m$. The adversary selects this path, corresponding to the schedule of crashes of subsequent groups on the path in consecutive rounds, prior the execution. This assures that in the course of the execution Consensus is not achieved within the first $\Theta(\sqrt{m/\log m})$ rounds, with probability at least $1 - 1/\log m$, thus also in terms of the expected number of rounds.

□

Note that the lower bound on the expected number of point-to-point messages is $\Omega(n)$ even for weaker Non-Adaptive$_{\mathrm{C}}$ adversaries, as each process has to receive a message to guarantee agreement in case of potential crashes, cf., [53] for references.

6.2 k-Chain-Ordered$_{\mathrm{C}}$ and k-Ordered$_{\mathrm{C}}$ adversaries

In this section we present KOCons - a modification of WACons specifically tailored to run against the k-Chain-Ordered$_{\mathrm{C}}$ adversary. Then, we also show that it is capable of running against a k-Ordered$_{\mathrm{C}}$ adversary.

The algorithm begins with electing a number of leaders in the Gather-Leaders

Algorithm 24: KOCons, pseudo-code for process v

```
WACons with Elect-Leader-Cons substituted by Gather-Leaders;
```

Algorithm 25: Gather-Leaders, pseudo-code for process v

```
initialize variable $n^*$;
$n^*$ := Count-Processes;
$i = \lfloor n/n^* \rfloor$;
coin := $\frac{k}{2^{i-1}n^*}$;
initialize list LEADERS to an empty list;
toss a coin with the probability coin of heads to come up;
if heads came up in the previous step then
    Propagate-Msg("$v$") to all other processes;
    add $v$ to list LEADERS;
end
fill in list LEADERS with elected leaders' identifiers from received messages;
return LEADERS;
```

Algorithm 26: Count-Processes, pseudo-code for process v

```
Propagate-Msg("$v$") to all other processes;
return the number of ID's heard ;
```

procedure. However, as the adversary models its pattern of crashes into k disjoint chains then we would like to elect approximately k leaders.

It may happen that the adversary significantly reduces the number of processes and hence, the leader election procedure is unsuccessful in electing an appropriate number of leaders. That is why we adjust the probability of success by approximating the size of the network before electing leaders. If the initial number of processes was n and the drop in the number of processes after estimating the size of the network was *not significant (less than half the number of the approximation)* then we expect to elect $\Theta(k)$ leaders. The approximation may not be exact and consistent in the whole system, yet it assures a monotonically decreasing approximation throughout the execution of KOCons.

Otherwise, if the number of processes was reduced by more than half, the probability of success is changed and the expected number of elected leaders is reduced. This helps to shorten executions of Leader-Consensus because a smaller number of leaders executes the protocol faster. In general if there are $\frac{n}{2^i}$ processes, we expect to elect $\Theta\left(\frac{k}{2^i}\right)$ leaders.

Elected leaders are expected to be placed uniformly in the adversary's order of crashes. If we look at a particular leader v, then he will be present in some chain k_i. What is more, his position within this chain is expected to be in the middle of k_i.

Leaders execute the Conditional-Consensus protocol Leader-Consensus. If they reach Consensus then they communicate this fact twice to the rest of the system. Hence, if the adversary wants to prolong the execution, then it must crash all leaders. Otherwise, the whole system would reach Consensus and end the protocol.

If leaders are placed uniformly in the adversary's order, then the adversary must preserve the pattern of crashes that it declared at first. In what follows, if there is a leader v that is placed in the middle of chain k_i, then half of the processes preceding v must also be crashed.

When the whole set of leaders is crashed then another group is elected and the process continues until the adversary spends all its possibilities of failing processes.

KOCons is a modification of WACons, hence, similarly as in the previous section, we analyze the expected time, assuming that the Leader-Consensus procedure is successful, yet we require that the algorithm satisfies termination, validity and agreement irrespectively of the success of Leader-Consensus.

In order to prove Theorem 18 we introduce the following Lemma.

Lemma 21. *Let L denote the number of leaders elected in* Gather-Leaders*, n^* be the number of operational processes, k denote the number of chains in the adversary's order and*

$l = \frac{k}{2^i}$ *for* $0 \le i \le \log k$. *Then* $l/4 < L < 3l/4$ *at least with probability* $1 - 2e^{-l/24}$.

Proof. Let X be a random variable such that $X = X_1 + \cdots + X_{n^*}$, where $X_1, \cdots, X_{n^*}$ are Poisson trials and

$$X_j = \begin{cases} 1 & \text{if process is chosen a leader,} \\ 0 & \text{otherwise.} \end{cases}$$

Since the number of operational processes is n^* and the probability of success is set to at most $\frac{l}{2n^*}$ in the Gather-Leaders procedure, we know that $\mu \le \mathbb{E}X = \mathbb{E}X_1 + \cdots + \mathbb{E}X_{n^*} = \frac{l}{2}$. To estimate the probability that the number of leaders lies within the expected interval we use the following Chernoff's inequalities:

1. $\mathbb{P}[X \ge (1+\delta)\mu] \le e^{-\mu\delta^2/3}$, for $0 < \delta \le 1$.

2. $\mathbb{P}[X \le (1-\epsilon)\mu] \le e^{-\mu\epsilon^2/2}$, for $0 < \epsilon < 1$.

Taking $\delta = \epsilon = \frac{1}{2}$, we have that $\mathbb{P}\left[X \ge \frac{3l}{4}\right] \le e^{-l/24}$ and $\mathbb{P}\left[X \le \frac{l}{4}\right] \le e^{-l/16}$. Hence $\mathbb{P}\left[X < \frac{3l}{4}\right] > 1 - e^{-l/24}$ and $\mathbb{P}\left[X > \frac{l}{4}\right] > 1 - e^{-l/16}$. In what follows $l/4 < L < 3l/4$ at least with probability $1 - 2e^{-l/24}$. □

We now proceed with the proof of Theorem 18

Theorem 18. *The following statements are true about* KOCons *against the k-Chain-Ordered$_C$ adversary:*
a) KOCons *reaches Consensus in the expected number of rounds equal* $\mathcal{O}(T_{LC}(k)\log(n/k))$, *satisfying termination, agreement and validity.*
b) *Expected work per process of* KOCons *is* $\frac{k}{n}\log\left(\frac{n}{k}\right)\sum_{i=0}^{\log k}\mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)$.
c) KOCons *reaches Consensus in the number of rounds equal* $\log\left(\frac{n}{k}\right)\sum_{i=0}^{\log k}\mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)$ *and with work per process* $\frac{k}{n}\log\left(\frac{n}{k}\right)\sum_{i=0}^{\log k}\mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)$ *at least with probability* $1 - e^{-\frac{k^{1/4}}{24}+\log\log k^2}$.

Proof. **a)** KOCons is structured as a repeat loop at the beginning of which a leader election procedure is executed. Gather-Leaders approximates the size of the network in the first place by a very simple procedure Count-Processes in order to adjust the coin values appropriately.

After the number of processes is counted it may happen that the adversary crashes some of the processes, so that their number drops significantly i.e. by more than half the number returned by Count-Processes. We will, however, firstly analyze the opposite situation when the drop is not significant to explain the basic situation and calculate the resulting time for the algoritm to succeed. Then we will proceed to the analysis of the situation when the adversary reduces the number of operational processes significantly and calculate a union bound over all such cases.

According to Lemma 21 Gather-Leaders returns $\Theta(k)$ leaders. In what follows the expected number of rounds required to reach Conditional-Consensus by Leader-Consensus is $\mathcal{O}(T_{LC}(k))$.

Note that the adversary has to crash all the leaders at some point of the Leader-Consensus protocol to prolong the execution. Otherwise, there would be a process that would reach Conditional-Consensus and inform all other processes about the consensus value.

According to the adversary's partial order there are initially k chains, where chain j has length l_j. An elected leader v belongs to one of the chains. We show that v is placed in the middle of that chain.

Let X_j be a random variable representing the position of leader j in a chain. We have that $\mathbb{E}X_j = \sum_{i=1}^{l_j} \frac{i}{l_j} = \frac{1}{l_j}\frac{(1+l_j)}{2}l_j = \frac{(1+l_j)}{2}$.

The crash of v implies that half of the processes forming the chain are also crashed. If at some other points of time, consecutively elected leaders will also belong to the same chain, then by simple induction we may conclude that this chain is expected to be entirely crashed after $\mathcal{O}(\log(n/k))$ steps in average.

In what follows if there are k chains and we expect to elect $\Theta(k)$ leaders, then after at most $\mathcal{O}(\log(n/k))$ steps this process ends, as the adversary runs out of all the possibilities of crashing. We estimate that each such step requires $\mathcal{O}(T_{LC}(k))$ time for running Leader-Consensus. Hence, the expected time for KOCons to reach Consensus is $\mathcal{O}(T_{LC}(k)\log(n/k))$.

Let us now proceed to the situation when the adversary crashes a significant number of processes (i.e., more than half of them). In such case procedure Gather-Leaders changes the probability of success for electing the leaders, as well as reduces the expected number of leaders that should be elected in the next iteration of the algorithm.

Once again, according to Lemma 21, we expect procedure Gather-Leaders to return $\Theta(k/2^i)$ leaders for some $0 \leq i \leq \log k$, since the number of such drops is bounded by $\log k$. Calculating the time required to reach Consensus over all cases gives us $\sum_{i=0}^{\log k} \mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\log\left(\frac{n}{2^i k}\right)\right) = \log\left(\frac{n}{k}\right)\sum_{i=0}^{\log k} \mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)$.

Altogether we have that KOCons requires $\log\left(\frac{n}{k}\right)\sum_{i=0}^{\log k} \mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)$ expected number of rounds to reach Consensus, where termination, validity and agreement are satisfied because of the same arguments as in Theorem 16.

b) Similarly as in the proof of Theorem 16, if there are n^* operational processes, and there were no significant drops in this number, we know that $\log\left(\frac{n}{k}\right)$ is the expected number of iterations of KOCons and each process becomes a leader in each iteration with probability $\frac{k}{n^*}$. The expected number of rounds in each iteration is $\mathcal{O}(T_{LC}(k))$, hence by applying Wald's equation ([85], Theorem 12.3) we have that $\frac{k}{n^*}\ \mathcal{O}(T_{LC}(k))$ is the expected work per process.

However assuming that significant drops may happen while Leader-Consensus procedure, we need to calculate a union bound over all the cases, which for each $0 \leq i \leq \log k$ gives us $\log\left(\frac{n}{2^i k}\right)$ expected number of iterations of KOCons and the probability that each process becomes a leader equal $\frac{k}{2^i \frac{n}{2^i}}$. The expected number of rounds in each iteration for a particular i is $\mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)$. Summing this altogether we have that the expected work per process is equal $\sum_{i=0}^{\log k} \frac{k}{n}\ \mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)\log\left(\frac{n}{2^i k}\right) = \frac{k}{n}\log\left(\frac{n}{k}\right)\sum_{i=0}^{\log k}\mathcal{O}\left(T_{LC}\left(\frac{k}{2^i}\right)\right)$.

c) The success of the algorithm depends on successful leader election through all the cases of significant drops in the number of operational processes. According to Lemma 21 for each $0 \leq i \leq \log k$ the probability that Leader-Consensus is successful in returning $\Theta\left(\frac{k}{2^i}\right)$ leaders is $1 - 2e^{-\frac{k}{2^i}\frac{1}{24}}$. We want to calculate the probability that Leader-Consensus is successful over all cases. Let A_i be an event that is was successful in the i'th case. We have that

$$\mathbb{P}\left(\bigcap_{i=1}^{\log k} A_i\right) = \prod_{i=1}^{\log k}\left(1 - 2e^{-\frac{k}{2^i}\frac{1}{24}}\right) \geq \prod_{i=1}^{\log k}\left(1 - 2e^{-\frac{k}{2^{\log k}}\frac{1}{24}}\right).$$

Since $2^{\log k} = e^{\log 2 \log k} = e^{\log k^{\log 2}} = k^{\log 2}$ and $\frac{1}{2} < \log 2 < 1$, thus

$$\prod_{i=1}^{\log k}\left(1 - 2e^{-\frac{k}{2^{\log k}}\frac{1}{24}}\right) \geq \prod_{i=1}^{\log k}\left(1 - 2e^{-\frac{k^{1/4}}{24}}\right) = \left(1 - 2e^{-\frac{k^{1/4}}{24}}\right)^{\log k}.$$

Using the Bernoulli inequality we continue having

$$\left(1 - 2e^{-\frac{k^{1/4}}{24}}\right)^{\log k} \geq 1 - 2\log k\ e^{-\frac{k^{1/4}}{24}} = 1 - e^{-\frac{k^{1/4}}{24} + \log\log k^2}.$$

Applying this result to points a) and b) finishes the proof.

□

Since, similarly as in the analysis of ChainPerformer we would like to instantiate procedure Leader-Consensus with algorithm SynRan from [14], we introduce the following fact, allowing us to sum the time complexity appropriately in Corollary 6.

Fact 13. *Let $n, k \in \mathbb{N}$, where $0 \leq k \leq n$. Then*

$$\sum_{m=0}^{\log k} \sqrt{\frac{k}{2^m \log\left(\frac{k}{2^m}\right)}} \log\left(\frac{n}{2^m k}\right) = c\sqrt{\frac{k}{\log k}} \log\left(\frac{n}{k}\right),$$

for some constant c.

Proof. We have that

$$\sum_{m=0}^{\log k} \sqrt{\frac{k}{2^m \log\left(\frac{k}{2^m}\right)}} \log\left(\frac{n}{2^m k}\right) \leq \log\left(\frac{n}{k}\right) \sum_{m=0}^{\log k} \sqrt{\frac{k}{2^m \log\left(\frac{k}{2^m}\right)}}.$$

Since $m \leq \log k$, then

$$\log\left(\frac{k}{2^m}\right) \geq \log\left(\frac{k}{2^{\log k}}\right) = \log(k^{1-\log 2}) = (1 - \log 2)\log k \geq \frac{1}{4}\log k,$$

because $2^{\log k} = e^{\log 2 \log k} = e^{\log k^{\log 2}} = k^{\log 2}$ and $\frac{1}{2} < \log 2 < 1$.
Hence

$$\begin{aligned}\log\left(\frac{n}{k}\right) \sum_{m=0}^{\log k} \sqrt{\frac{k}{2^m \log\left(\frac{k}{2^m}\right)}} &\leq 2\log\left(\frac{n}{k}\right) \sum_{m=0}^{\log k} \sqrt{\frac{k}{2^m \log k}} \\ &\leq 2\sqrt{\frac{k}{\log k}} \log\left(\frac{n}{k}\right) \sum_{m=0}^{\log k} 2^{-m/2} \qquad \leq c\sqrt{\frac{k}{\log k}} \log\left(\frac{n}{k}\right).\end{aligned}$$

□

Corollary 6. *Instantiating* Leader-Consensus *with* SynRan *from [14] results in:*
a) $\mathcal{O}\left(\sqrt{\frac{k}{\log k}} \log(n/k)\right)$ *expected number of rounds to reach Consensus by* KOCons.
b) *Expected work per process equal* $\frac{k}{n} \log\left(\frac{n}{k}\right) \mathcal{O}\left(\sqrt{\frac{k}{\log k}} \log(n/k)\right)$.
c) $\mathcal{O}\left(\sqrt{\frac{k}{\log k}} \log(n/k)\right)$ *rounds to reach Consensus by* KOCons *with probability* $1 - e^{-\frac{k^{1/4}}{24} + \log\log k^2}$.

6.2.1 KOCons against the adversary limited by an arbitrary partial order

Similarly, as in Section 3.3.3, let us consider the adversary that is limited by an **arbitrary** partial order relation $\succ$ on the set of all processes. Two elements in this partially ordered set are *incomparable* if neither $x \succ y$ nor $y \succ x$ hold. Translating this into our model, the adversary may crash incomparable elements in any sequence during the execution of the algorithm. We assume that crashes forced by the adversary are constrained by some partial order P.

Theorem 19. KOCons *reaches Consensus in expected* $\mathcal{O}(T_{LC}(k)\log(n/k))$ *number of rounds, against the* k*-Ordered*$_C$ *adversary, satisfying termination, agreement and validity.*

Proof. We assume that crashes forced by the adversary are constrained by some partial order $\succ$. Let us recall the following lemma.

Lemma 22. *(Dilworth's theorem [39]) In a finite partial order, the size of a maximum anti-chain is equal to the minimum number of chains needed to cover all elements of the partial order.*

The k-Ordered$_{\text{C}}$ adversary is constrained by an arbitrary order of thickness k. The adversary by choosing some f processes to be crashed cannot increase the size of the maximal anti-chain. Thus using Lemma 22 we consider the coverage of the crash-prone processes by at most k disjoint chains and any dependencies between chains' elements create additional constraints to the adversary comparing to the k-Chain-Ordered$_{\text{C}}$ one. Hence we fall into the case concluded in Theorem 18 and that completes the proof.

□

6.2.2 Lower bound

We finish with the lower bound for reaching Consensus against the k-Ordered$_{\text{C}}$ adversary.

Theorem 20. *For any reliable randomized algorithm solving Consensus in a message-passing model and any integer* $0 < k \leq f$*, there is a* k*-Ordered adversary that can force the algorithm to run in* $\Omega(\sqrt{k/\log k})$ *expected number of rounds.*

Proof. The adversary takes the order consisting of n/k independent chains, and follows the analysis for Strongly-Adaptive$_{\text{C}}$ adversaries in [14], with the following modification: if the analysis enforces crashing a process v, the adversary crashes the whole chain to which v belongs. Since there are k chains and $k-1$ possible chain-crashes, by replacing n with k in the formula obtained in [14] we get $\Omega(\sqrt{k/\log k})$ expected number of rounds.

□

6.3 Conclusions

In this chapter, we showed time efficient randomized consensus protocols against the Weakly-Adaptive$_\mathrm{C}$, Non-Adaptive$_\mathrm{C}$ and Ordered$_\mathrm{C}$ adversaries generating crashes. We proved that all these classes of Constrained adaptive adversaries are weaker than the Strongly-Adaptive$_\mathrm{C}$ one. Our results also extend the state-of-the-art of the study of popular Non-Adaptive$_\mathrm{C}$ adversaries.

Three main open directions emerge from this chapter. One is to improve the message complexity of proposed algorithms and make them resistant to (rarely expected, but possible) very long executions resulting from unsuccessful probabilistic events. Another open direction could pursue a study of complexities of other important distributed problems and settings against Weakly-Adaptive$_\mathrm{C}$ and Ordered$_\mathrm{C}$ adversaries, which are more realistic than the Strongly-Adaptive$_\mathrm{C}$ adversary and more general than the Non-Adaptive$_\mathrm{C}$ adversary, commonly used in the literature. Finally, there is space for proposing and studying other, intermediate types of adversaries, including further study of Round-Delay adversaries, cf., Section 2.1 and adversaries tailored for dynamic distributed and parallel computing [69].

Chapter 7

Summary

In this thesis, we studied two fundamental problems in distributed computing i.e., the Do-All problem and the Consensus problem, in presence of adversarial crashes. We analyzed both of them against adaptive adversaries constrained by several classes of partial orders, which represent patterns of crashes. Furthermore, we analyzed the Do-All problem with different assumptions about tasks, including arbitrary length tasks and tasks, with dependencies between each other.

The results of Chapter 3 provide a hierarchy of Ordered and Round-Delay adversaries, which prove a separation of results for different scenarios. A subset of results, regarding Ordered adversaries, yields a conclusion that the difficulty of the problem grows with the size of the maximal anti-chain of the partial order of the adversary. Results for Round-Delay adversaries prove that delaying the Strongly-Adaptive adversary's decisions by just a single round, eases the problem significantly.

In Chapter 4, we considered the Do-All problem for arbitrary length tasks, distinguishing whether preemption is available or not. Results from this chapter showed that there is a distinction between the two models. Additionally, we showed that while randomization helps with improving the efficiency of algorithms against Non-Adaptive adversaries for the preemptive setting, it is not helpful against the Strongly-Adaptive adversary.

Building on the results for arbitrary length preemptive tasks, we suggested an algorithm solving Do-All with dependent tasks forming partially ordered sets of chains in Chapter 5. The result was further extended to tree-shaped partial orders of tasks. Results from this chapter were complemented with a heuristic algorithm for an arbitrary partial order of tasks and considerations about a mitigation of task dependencies in the form of an OR policy, where performing just a single child of a task opens access to its parent.

Chapter 6 contains results regarding the Consensus problem in a synchronous message-passing system against the Weakly-Adaptive$_{\mathrm{C}}$ adversary and the k-Ordered$_{\mathrm{C}}$ adversary, which constitute a novel framework for evaluating the complexity of the problem. Moreover, we commented on the message complexity in such setting and proved appropriate lower bounds for considered adversarial scenarios.

Apart from the specific open questions stated in the end of each chapter, perhaps there are two most interesting open directions emerging from this thesis. The first concerns further extensions of the theory of scheduling on a multiple-access channel, by examining different features considered in scheduling literature. The second regards exploring different problems in the field of distributed computing under intermediate types of adversaries and, in general, direct research on fault-tolerance to more realistic models.

Bibliography

[1] Norman Abramson. Development of the alohanet. *IEEE Transactions on Information Theory*, 31(2):119–123, September 2006.

[2] Eugene S. Amdur, Samuel M. Weber, and Vassos Hadzilacos. On the message complexity of binary byzantine agreement under crash failures. *Distributed Computing*, 5(4):175–186, April 1992.

[3] James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, May 1998.

[4] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, September 2003.

[5] James Aspnes, Hagit Attiya, and Keren Censor. Randomized consensus in expected $\mathcal{O}(n \log n)$ individual work. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 325–334, New York, NY, USA, 2008. ACM.

[6] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11:441–461, 1990.

[7] James Aspnes and Orli Waarts. Randomized consensus in expected $\mathcal{O}(n \log^2 n)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.

[8] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):20:1–20:26, November 2008.

[9] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, Inc., USA, 2004.

[10] Yonatan Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 209–218, New York, NY, USA, 1997. ACM.

[11] Yonatan Aumann and Michael A. Bender. Efficient low-contention asynchronous consensus with the value-oblivious adversary scheduler. *Distributed Computing*, 17:191–207, March 2005.

[12] Baruch Awerbuch, Andréa W. Richa, and Christian Scheideler. A jamming-resistant MAC protocol for single-hop wireless networks. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 45–54. ACM, 2008.

[13] Evripidis Bampis, Dimitrios Letsios, and Giorgio Lucarelli. A note on multiprocessor speed scaling with precedence constraints. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 138–142, New York, NY, USA, 2014. ACM.

[14] Ziv Bar-Joseph and Michael Ben-Or. A tight lower bound for randomized synchronous consensus. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 193–199, New York, NY, USA, 1998. ACM.

[15] 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.

[16] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Maxwell Young. How to scale exponential backoff: Constant throughput, polylog access attempts, and robustness. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 636–654. SIAM, 2016.

[17] Gabriel Bracha and Ophir Rachman. Randomized consensus in expected $\mathcal{O}(n^2 \log n)$ operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, WDAG '91, pages 143–150, Berlin, Heidelberg, 1992. Springer-Verlag.

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

[19] Philipp Brandes, Marcin Kardas, Marek Klonowski, Dominik Pająk, and Roger Wattenhofer. Approximating the size of a radio network in beeping model. In Jukka Suomela, editor, *Structural Information and Communication Complexity - 23rd International Colloquium, SIROCCO 2016, Helsinki, Finland, July 19-21, 2016, Revised Selected Papers*, volume 9988 of *Lecture Notes in Computer Science*, pages 358–373, 2016.

[20] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'96, pages 166–175, New York, NY, USA, 1996. ACM.

[21] Chandra Chekuri and Rajeev Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(1):29 – 38, 1999.

[22] Binbin Chen, Ziling Zhou, and Haifeng Yu. Understanding RFID counting protocols. In Sumi Helal, Ranveer Chandra, and Robin Kravets, editors, *The 19th Annual International Conference on Mobile Computing and Networking, MobiCom'13, Miami, FL, USA, September 30 - October 04, 2013*, pages 291–302. ACM, 2013.

[23] Bogdan S Chlebus. *Randomized communication in radio networks, a chapter. In: Pardalos, P.M., Rajasekaran, S., Reif, J.H., Rolim, J.D.P. (eds.), Handbook on Randomized Computing.*, volume 1. Kluwer Academic Publisher, 2001.

[24] Bogdan S. Chlebus, Roberto De Prisco, and Alex A. Shvartsman. Performing tasks on synchronous restartable message-passing processors. *Distributed Computing*, 14(1):49–64, January 2001.

[25] Bogdan S. Chlebus, Leszek Gąsieniec, Dariusz R. Kowalski, and Alexander A. Shvartsman. Bounding work and communication in robust cooperative computation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 295–310, London, UK, 2002. Springer-Verlag.

[26] Bogdan S. Chlebus, Karol Gołąb, and Dariusz R. Kowalski. Broadcasting spanning forests on a multiple access channel. *Theory of Computing Systems*, 36:711–733, 2003.

[27] Bogdan S. Chlebus and Dariusz R. Kowalski. Randomization helps to perform independent tasks reliably. *Random Structures and Algorithms*, 24(1):11–41, 2004.

[28] Bogdan S. Chlebus and Dariusz R. Kowalski. Robust gossiping with an application to consensus. *Journal of Computer and System Sciences*, 72(8):1262–1281, December 2006.

[29] Bogdan S. Chlebus and Dariusz R. Kowalski. Time and communication efficient consensus for crash failures. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, pages 314–328, 2006.

[30] Bogdan S. Chlebus and Dariusz R. Kowalski. Locally scalable randomized consensus for synchronous crash failures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 290–299, New York, NY, USA, 2009. ACM.

[31] Bogdan S. Chlebus, Dariusz R. Kowalski, and Andrzej Lingas. Performing work in broadcast networks. *Distributed Computing*, 18(6):435–451, 2006.

[32] Bogdan S. Chlebus, Dariusz R. Kowalski, and Michal Strojnowski. Fast scalable deterministic consensus for crash failures. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 111–120, New York, NY, USA, 2009. ACM.

[33] Bogdan S. Chlebus, Roberto De Prisco, and Alexander A. Shvartsman. Performing tasks on synchronous restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.

[34] Benny Chor, Michael Merritt, and David B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, July 1989.

[35] Fabián A. Chudak and David B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *Journal of Algorithms*, 30(2):323 – 343, 1999.

[36] Andrea E. F. Clementi, Angelo Monti, and Riccardo Silvestri. Optimal f-reliable protocols for the do-all problem on single-hop wireless networks. In *Proceedings of*

the 13th International Symposium on Algorithms and Computation, ISAAC '02, pages 320–331, London, UK, UK, 2002. Springer-Verlag.

[37] Ed G. Coffman, Jr. and Michael R. Garey. Proof of the 4/3 conjecture for preemptive vs. nonpreemptive two-processor scheduling. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 241–248, New York, NY, USA, 1991. ACM.

[38] Roberto De Prisco, Alain Mayer, and Moti Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 161–172, New York, NY, USA, 1994. ACM.

[39] Robert P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

[40] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM*, 32(1):191–204, January 1985.

[41] Cynthia Dwork, Joseph Y. Halpern, and Orli Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, 27(5):1457–1491, 1998.

[42] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, August 1997.

[43] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183 – 186, 1982.

[44] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[45] Zvi Galil, Alain Mayer, and Moti Yung. Resolving message complexity of byzantine agreement and beyond. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 724–733, Washington, DC, USA, 1995. IEEE Computer Society.

[46] Robert G. Gallager. A perspective on multiaccess channels. *IEEE Transactions on Information Theory*, 31:124–142, 1985.

[47] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement in t + 1 rounds. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 31–41, New York, NY, USA, 1993. ACM.

[48] Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the complexity of asynchronous gossip. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 135–144, New York, NY, USA, 2008. ACM.

[49] Chryssis Georgiou, Dariusz R. Kowalski, and Alexander A. Shvartsman. Efficient gossip and robust distributed computation. *Theoretical Compututer Science*, 347(1-2):130–166, November 2005.

[50] Chryssis Georgiou and Alexander Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity.* Springer Publishing Company, Incorporated, 2008.

[51] Chryssis Georgiou and Alexander A. Shvartsman. Cooperative Task-Oriented Computing: Algorithms and Complexity. *Synthesis Lectures on Distributed Computing Theory*, 2(2):1–167, 2011.

[52] Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the message complexity of indulgent consensus. In Andrzej Pelc, editor, *Distributed Computing*, pages 283–297, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[53] Seth Gilbert and Dariusz R. Kowalski. Distributed agreement with optimal communication complexity. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 965–977, 2010.

[54] Albert G. Greenberg and Schmuel Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *Journal of the ACM*, 32(3):589–596, July 1985.

[55] Vassos Hadzilacos and Joseph Y. Halpern. Message-optimal protocols for byzantine agreement. *Mathematical systems theory*, 26(1):41–102, Jan 1993.

[56] Wassily Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[57] Dan Holtby, Bruce M. Kapron, and Valerie King. Lower bound for scalable byzantine agreement. *Distributed Computing*, 21(4):239–248, October 2008.

[58] Johann Hurink and Sigrid Knust. List scheduling in a parallel machine environment with precedence constraints and setup times. *Operations Research Letters*, 29(5):231 – 239, 2001.

[59] Tomasz Jurdziński, Mirosław Kutyłowski, and Jan Zatopiański. Efficient algorithms for leader election in radio networks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 51–57, New York, NY, USA, 2002. ACM.

[60] Paris C. Kanellakis and Alex A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4):201–217, 1992.

[61] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pages 87–98, Washington, DC, USA, 2006. IEEE Computer Society.

[62] Marek Klonowski, Dariusz R. Kowalski, and Jarosław Mirek. Ordered and delayed adversaries and how to work against them on a shared channel. *Distributed Computing*, Sep 2018.

[63] Marek Klonowski, Dariusz R. Kowalski, Jarosław Mirek, and Prudence W. H. Wong. Fault-tolerant parallel scheduling of arbitrary length jobs on a shared channel. In Leszek Antoni Gąsieniec, Jesper Jansson, and Christos Levcopoulos, editors, *Fundamentals of Computation Theory*, pages 306–321, Cham, 2019. Springer International Publishing.

[64] Marek Klonowski, Dariusz R. Kowalski, Jarosław Mirek, and Prudence W. H. Wong. Performing partially ordered sets of jobs on a mac in presence of adversarial crashes. In *International Symposium on Network Computing and Applications (NCA 2019)*, To appear.

[65] Marek Klonowski and Dominik Pajak. Electing a leader in wireless networks quickly despite jamming. In Guy E. Blelloch and Kunal Agrawal, editors, *Proceedings of the*

27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, pages 304–312. ACM, 2015.

[66] Janos Komlos and Albert Greenberg. An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Transactions on Information Theory*, 31(2):302–306, September 2006.

[67] Dariusz R. Kowalski. On selection problem in radio networks. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 158–166, New York, NY, USA, 2005. ACM.

[68] Dariusz R. Kowalski and Jarosław Mirek. On the complexity of fault-tolerant consensus. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems*, pages 19–31, Cham, 2019. Springer International Publishing.

[69] Dariusz R. Kowalski and Miguel A. Mosteiro. Polynomial counting in anonymous dynamic networks with applications to anonymous dynamic algebraic computations. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 156:1–156:14, 2018.

[70] Dariusz R. Kowalski and Alex A. Shvartsman. Performing work with asynchronous processors: Message-delay-sensitive bounds. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 265–274, New York, NY, USA, 2003. ACM.

[71] Dariusz R. Kowalski, Prudence W. Wong, and Elli Zavou. Fault tolerant scheduling of tasks of two sizes under resource augmentation. *Journal of Scheduling*, 20(6):695–711, December 2017.

[72] Eyal Kushilevitz and Yishay Mansour. An $\omega(d \log(n/d))$ lower bound for broadcast in radio networks. *SIAM Journal on Computing*, 27(3):702–712, June 1998.

[73] Eugene L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19(5):544–546, January 1973.

[74] Eugene L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. In B. Alspach, P. Hell, and D.J. Miller, editors, *Algorithmic Aspects of Combinatorics*, volume 2 of *Annals of Discrete Mathematics*, pages 75 – 90. Elsevier, 1978.

[75] Eugene L. Lawler. Preemptive scheduling of. precedence-constrained jobs on parallel machines. In M. A. H. Dempster, Jan K. Lenstra, and Alexander H. G. Rinnooy Kan, editors, *Deterministic and Stochastic Scheduling*, pages 101–123, Dordrecht, 1982. Springer Netherlands.

[76] Jan K. Lenstra and Alexander H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, February 1978.

[77] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* CRC Press, Inc., Boca Raton, FL, USA, 2004.

[78] Joseph Leung and Gilbert H. Young. Minimizing total tardiness on a single machine with precedence constraints. *ORSA Journal on Computing*, 2(4):346–352, 1990.

[79] Michael C Loui and Hosame H Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research*, 4(163-183):31, 1987.

[80] Giorgio Lucarelli, Abhinav Srivastav, and Denis Trystram. From preemptive to non-preemptive scheduling using rejections. In Thang N. Dinh and My T. Thai, editors, *Computing and Combinatorics*, pages 510–519, Cham, 2016. Springer International Publishing.

[81] Irene N. Lushchakova. Two machine preemptive scheduling problem with release dates, equal processing times and precedence constraints. *European Journal of Operational Research*, 171(1):107 – 122, 2006.

[82] Charles U. Martel. Maximum finding on a multiple access broadcast network. *Information Processing Letters*, 52(1):7–13, October 1994.

[83] Robert McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, October 1959.

[84] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[85] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, New York, NY, USA, 2005.

[86] Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.

[87] Kirk Pruhs, Rob van Stee, and Patchrawat Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43(1):67–80, Jul 2008.

[88] Michael O. Rabin. Randomized byzantine generals. *24th Annual Symposium on Foundations of Computer Science (FOCS 1983)*, pages 403–409, 1983.

[89] Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. Competitive and fair medium access despite reactive jamming. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 507–516. IEEE Computer Society, 2011.

[90] Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. An efficient and fair MAC protocol robust to reactive interference. *IEEE/ACM Transactions on Networking*, 21(3):760–771, 2013.

[91] Peter Robinson, Christian Scheideler, and Alexander Setzer. Breaking the $\Omega(\sqrt{n})$ barrier: Fast consensus under a late adversary. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 173–182, 2018.

[92] Martin Skutella and Marc Uetz. Scheduling precedence-constrained jobs with stochastic processing times on parallel machines. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 589–590, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[93] Dan E Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM Journal on Computing*, 15(2):468–477, May 1986.

[94] Gerhard J. Woeginger. On the approximability of average completion time scheduling under precedence constraints. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 887–897, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.