On the Complexity of Fault-Tolerant Consensus

Dariusz R. Kowalski[1,2] and Jarosław Mirek[2(✉)]

[1] School of Computer and Cyber Sciences, Augusta University, Augusta, USA
[2] Department of Computer Science, University of Liverpool, Liverpool, UK
{D.Kowalski,J.Mirek}@liverpool.ac.uk

Abstract. 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* adversary, who has to fix, in advance, the set of f crash-prone processes, and a *k-Chain-Ordered* 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* adversaries model attacks and *Non-Adaptive* 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 better the crash-tolerant consensus 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* adversaries to hold also for (syntactically) weaker *Non-Adaptive* adversaries. Together with the consensus algorithm against *Weakly-Adaptive* adversaries (which automatically translates to the *Non-Adaptive* adversaries), these results extend the state-of-the-art of the popular class of *Non-Adaptive* adversaries, in particular, the result of Chor, Meritt and Shmoys [7], and prove separation gap between *Constrained* adversaries (including *Non-Adaptive* ones) and *Strongly-Adaptive* adversaries, analyzed by Bar-Joseph and Ben-Or [3] and others.

1 Introduction

We study the problem of consensus in synchronous message passing distributed systems. There are n processes, out of which at most f can crash. Each process is initialized with a binary input value, and the goal is to agree on a common value (from the input values) by all processes. 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; and *termination:* each process eventually decides, unless it crashes. In case of randomized

Supported by the Polish National Science Center (NCN) grant UMO-2017/25/B/ST6/02553.

M. F. Atig and A. A. Schwarzmann (Eds.): NETYS 2019, LNCS 11704, pp. 19–31, 2019.
https://doi.org/10.1007/978-3-030-31277-0_2

solutions, the specification of consensus needs to be reformulated, which can be done in various ways (cf., [2]). We consider a classic reformulation in which validity and agreement are required to hold for every execution, while termination 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.

Randomization has been used in consensus algorithms for various kinds of failures specified by adversarial models, see [1,2]. 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 [7,9,18]. Feasibility of achieving small upper bounds on performance of algorithms solving consensus in a given distributed environment depends on the power of adversaries inflicting failures.

1.1 Previous and Related Work

Consensus is one of the fundamental problems in distributed computing, with a rich history of research done in various settings and systems, cf., [2]. Recently its popularity grew even further due to applications in emerging technologies such as blockchains. Below we present only a small digest of literature closely related with the setting considered in this work.

Consensus is solvable in synchronous systems with processes prone to crashing, although time $f+1$ is required [10] and sufficient [12] in case of deterministic solutions. Chor, Meritt and Shmoys [7] 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 [3] 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 [11] 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 [17] 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 probability 1.

Bracha and Toueg [5] 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 [4] gave the first randomized algorithm solving consensus in the asynchronous message passing model under the assumption that the majority of processes are non-faulty.

The consensus problem has been recently considered against different adversarial scenarios. Robinson, Scheideler and Setzer [19] 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.

1.2 Our Results

Table 1. Time complexity of solutions for the consensus problem against different adversaries. Formulas with * are presented in this paper.

		Strongly-Adaptive	*Weakly-Adaptive* and *Non-Adaptive*	*k-Chain-Ordered*
Randomized	Upper bound	$\mathcal{O}\left(\sqrt{\frac{n}{\log n}}\right)$ [3]	$\mathcal{O}\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ *	$\mathcal{O}\left(\sqrt{\frac{k}{\log k}}\log(n/k)\right)$ *
	Lower bound	$\Omega\left(\sqrt{\frac{n}{\log n}}\right)$ [3]	$\Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ *	$\Omega\left(\sqrt{\frac{k}{\log k}}\right)$ *
Deterministic	Upper bound	$f+1$ [12]		
	Lower bound	$f+1$ [10]		

We analyze the consensus problem against restricted adaptive adversaries. The motivation is that a *Strongly-Adaptive* adversary, typically used for analysis of randomized consensus algorithms, may not be very realistic; for instance, in practice some 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 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, analyzing 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, interesting theoretical findings about the performance of consensus algorithms.

Table 1 presents time complexities of solutions for the consensus problem against different adversaries. Results for the *Strongly-Adaptive* adversary and for deterministic algorithms are known (see Sect. 1.1), while the other ones are delivered in this work. We design and analyze a randomized algorithm that reaches consensus in expected $\mathcal{O}\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ rounds against any *Weakly-Adaptive* 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 expected number of rounds.

The lower bound could be also generalized to hold against the (syntactically) weaker *Non-Adaptive* adversaries, therefore all the results concerning *Weakly-Adaptive* adversaries delivered in this paper hold for *Non-Adaptive* adversaries as well. This extends the state-of-the-art of the study of *Non-Adaptive* adversaries

done in high volume of previous work, cf., [6,7,13], specifically, an $O(1)$ expected time algorithm of Chor et al. [7] 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 $\Omega\left(\frac{f}{\sqrt{n \log n}}\right)$ [3] on the expected number of rounds of any consensus algorithm against a *Strongly-Adaptive* adversary crashing at most f processes, *our result shows a separation between the two important classes of adversaries – Non-Adaptive and Strongly-Adaptive – for the consensus problem, which is one of the most fundamental problems in distributed computing.*

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. The algorithm reaches consensus in $\mathcal{O}\left(\sqrt{\frac{k}{\log k}} \log(n/k)\right)$ rounds in expectation. Additionally, we show a lower bound $\Omega\left(\sqrt{\frac{k}{\log k}}\right)$ for the problem against a *k-Ordered* adversary. Finally, we show that this solution is capable of running against an arbitrary partial order with a maximal anti-chain of size k. Similarly to results for the *Weakly-Adaptive* adversary, formulas obtained for *Ordered* adversaries separate them from *Strongly-Adaptive* ones.

2 Model

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 [3], 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.

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 will crash, it can decide which subset of messages will reach their recipients.

In the context of the adversaries in this paper 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* adversary and hence cannot be crashed.
– Non-faulty - processes that survived until the end of the algorithm.

- *Strongly-Adaptive and Weakly-Adaptive adversaries.* The only restriction for the *Strongly-Adaptive* 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* adversary is consistent with the *Strongly-Adaptive* adversary, because it could simulate the algorithm before its execution and decide on choosing the most convenient subset of processes.

- *k-Chain-Ordered and k-Ordered adversaries.* The notion of a *k-Chain-Ordered* adversary originates from partial order relations, hence appropriate notions and definitions translate straightforwardly. The relation of our particular interest while considering partially ordered adversaries is the precedence relation. Precisely, 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. Consequently a subset of processes where every pair of processes is comparable is called a chain. On the other hand a subset of processes where no two different processes are comparable is called an anti-chain.

It is convenient to think about the partial order of the adversary from a Hasse diagram perspective. The notion of chains and anti-chains seems to be intuitive when graphically presented, e.g., a chain is a pattern of consecutive crashes that may occur while an anti-chain gives the adversary freedom to crash in any order due to non-comparability of processes.

Formally, the *k-Chain-Ordered* 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* adversary.

We refer to a wider class of adversaries in this paper, constrained by an arbitrary partial order, as *Ordered* adversaries. What is more, adversaries having additional limitations, apart from the possible number of crashes (i.e. all described in this paper but the *Strongly-Adaptive* adversary), will be called *Constrained* adversaries. Note that *Ordered* adversaries are also restricted by the number of possible crashes f they may enforce.

- *Non-Adaptive adversaries.* The *Non-Adaptive* 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.

Consensus Problem. 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 the presence of the adversary, capable of crashing processes. We require any consensus protocol to fulfill 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 only possible decision value.
– Termination: all non-faulty processes decide with probability 1.

We follow typical assumption that the first two requirements must hold in *any* execution, while termination should be satisfied with probability 1.

Complexity Measure and Algorithmic Tools. The main complexity measure used to benchmark the consensus problem is the number of rounds by which all non-faulty processes decide on a common value.

Throughout the paper we use black-box fashioned procedures that allow us to structure the presentation better. We now briefly describe their properties and later refer to them in the algorithms' analysis. Details could be found in the full version of this paper [15].

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 [3], and we refer the reader to the details therein. In particular, to Lemma 4.2 [3], 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.

3 *Weakly-Adaptive* Adversary

In this section we consider the fundamental result i.e. ALGORITHM A that consists of two main components - a leader election procedure, and a reliable consensus protocol. We combine them together in an appropriate way (cf., Fig. 1), in order to reach consensus against a *Weakly-Adaptive* adversary.

Algorithm 1: ALGORITHM A, pseudocode for process v

```
1  initialize list LEADERS to an empty list;
2  decided := false;
3  value := x_v;
4  repeat
5      LEADERS := ELECT-LEADER;
6      if LEADERS contains v then
7          (decided, value) := LEADER-CONSENSUS(T_LC(|LEADERS|), value) ;
8          if decided then
9              execute PROPAGATE-MSG(value) twice;
10         end
11         else
12             if heard the same consensus value CV_w twice from some process w then
13                 value := CV_w;
14                 decided = true;
15             end
16             if heard consensus value CV_w once from some process w then
17                 value := CV_w;
18             end
19         end
20     end
21     else
22         idle for T_LC rounds;
23         if heard the same consensus value CV_w twice from some process w then
24             value := CV_w;
25             decided = true;
26         end
27         if heard consensus value CV_w once from some process w then
28             value := CV_w;
29         end
30     end
31     clear list LEADERS;
32 until decided;
```

Algorithm 2: ELECT-LEADER, pseudocode for process v

```
1 coin := 1/(n−f);
2 initialize list LEADERS to an empty list;
3 toss a coin with the probability coin of heads to come up;
4 if heads came up in the previous step then
5 |   PROPAGATE-MSG("v") to all other processes;
6 |   add v to list LEADERS;
7 end
8 fill in list LEADERS with elected leaders' identifiers from received messages;
9 return LEADERS;
```

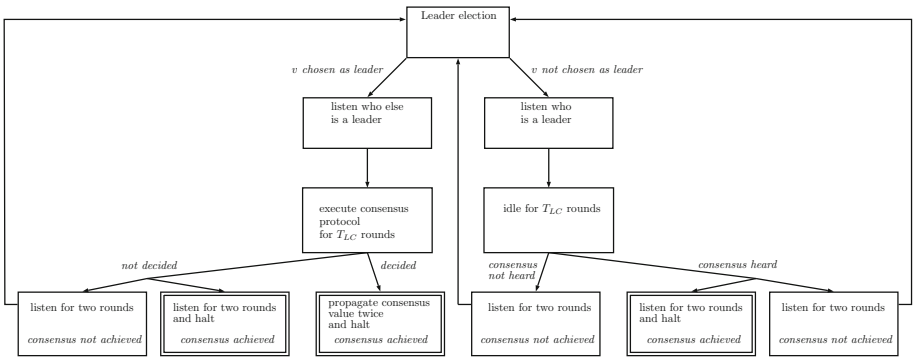

Fig. 1. ALGORITHM A flow diagram for process v.

ALGORITHM A 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 certain probability. If they do so, this fact is propagated to all processes via PROPAGATE-MSG so that all processes that were not in the leaders group, know about small 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 1 from the perspective of some process v. At the beginning of the protocol every process takes part in ELECT-LEADER procedure and process v tosses a coin with probability of success equal $\frac{1}{n-f}$ and either is 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*, what happens with certain 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 this 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 Algorithm A 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.

Theorem 1. Algorithm A *reaches consensus in the expected number of rounds equal* $\mathcal{O}\left(T_{LC}\left(\frac{n}{n-f}\right)\right)$*, satisfying termination, agreement and validity.*

Corollary 1. *Instantiating* Leader-Consensus *with* SynRan *from [3] results in* $\mathcal{O}\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$ *expected rounds to reach consensus by* Algorithm A.

Theorem 2. *The expected number of rounds of any consensus protocol running against a Weakly-Adaptive or a Non-Adaptive adversary causing up to f crashes is* $\Omega\left(\sqrt{\frac{n}{(n-f)\log(n/(n-f))}}\right)$.

4 *k-Chain-Ordered* and *k-Ordered* Adversaries

In this section we present Algorithm C - a modification of Algorithm A specifically tailored to run against the *k-Chain-Ordered* adversary. Then we also show that it is capable of running against a *k-Ordered* adversary.

Algorithm 3: ALGORITHM C, pseudocode for process v

```
1 ALGORITHM A with ELECT-LEADER substituted by GATHER-LEADERS;
```

The algorithm begins with electing a number of leaders in GATHER-LEADERS. 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.

Algorithm 4: GATHER-LEADERS, pseudocode for process v

```
1  initialize variable n*;
2  n* := COUNT-PROCESSES;
3  i = ⌊n/n*⌋;
4  coin := k / (2^(i-1) n*);
5  initialize list LEADERS to an empty list;
6  toss a coin with the probability coin of heads to come up;
7  if heads came up in the previous step then
8  |  PROPAGATE-MSG("v") to all other processes;
9  |  add v to list LEADERS;
10 end
11 fill in list LEADERS with elected leaders' identifiers from received messages;
12 return LEADERS;
```

Algorithm 5: COUNT-PROCESSES, pseudocode for process v

```
1 PROPAGATE-MSG("v") to all other processes;
2 return the number of ID's heard ;
```

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 small 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.

Theorem 3. Algorithm C *reaches consensus in the expected number of rounds equal* $\mathcal{O}(T_{LC}(k)\log(n/k))$*, satisfying termination, agreement and validity.*

Corollary 2. *Instantiating* Leader-Consensus *with* SynRan *from [3] results in* $\mathcal{O}\left(\sqrt{\frac{k}{\log k}}\log(n/k)\right)$ *expected number of rounds to reach consensus by* Algorithm C.

4.1 Algorithm C Against the Adversary Limited by an Arbitrary Partial Order

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. Let us recall the following lemma.

Lemma 1 *(Dilworth's theorem [8]). 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.*

Combining Lemma 1 with Theorem 3 and its instantiated form in Corollary 2, we obtain the following.

Theorem 4. Algorithm C *reaches consensus in expected* $\mathcal{O}(T_{LC}(k)\log(n/k))$ *number of rounds, against the k-Ordered adversary, satisfying termination, agreement and validity.*

We finish with the lower bound for reaching consensus against the *k-Ordered* adversary.

Theorem 5. *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.*

5 Conclusions and Open Problems

In this work we showed time efficient randomized consensus against the *Weakly-Adaptive*, *Non-Adaptive* and *Ordered* adversaries generating crashes. We proved that all these classes of *Constrained* adaptive adversaries are weaker than the *Strongly-Adaptive* one. Our results also extend the state-of-the-art of the study of popular *Non-Adaptive* adversaries.

Three main open directions emerge from this work. 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* and *Ordered* adversaries, which are more realistic than the *Strongly-Adaptive* one and more general than the *Non-Adaptive* one, commonly used in the literature. Finally, there is a scope of proposing and studying other intermediate types of adversaries, including further study of recently proposed delayed adversaries [14] and adversaries tailored for dynamic distributed and parallel computing [16].

References

1. Aspnes, J.: Randomized protocols for asynchronous consensus. Distrib. Comput. **16**(2–3), 165–175 (2003)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons Inc., USA (2004)
3. Bar-Joseph, Z., Ben-Or, M.: A tight lower bound for randomized synchronous consensus. In: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1998, New York, NY, USA, pp. 193–199. ACM (1998)
4. Ben-Or, M.: 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 1983, New York, NY, USA, pp. 27–30. ACM (1983)
5. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM **32**(4), 824–840 (1985)
6. Chlebus, B.S., Kowalski, D.R.: Locally scalable randomized consensus for synchronous crash failures. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009, New York, NY, USA, pp. 290–299. ACM (2009)
7. Chor, B., Merritt, M., Shmoys, D.B.: Simple constant-time consensus protocols in realistic failure models. J. ACM **36**(3), 591–614 (1989)
8. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Ann. Math. **51**(1), 161–166 (1950)
9. Feldman, P., Micali, S.: An optimal probabilistic protocol for synchronous byzantine agreement. SIAM J. Comput. **26**(4), 873–933 (1997)
10. Fischer, M.J., Lynch, N.A.: A lower bound for the time to assure interactive consistency. Inf. Process. Lett. **14**(4), 183–186 (1982)
11. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)

12. Garay, J.A., Moses, Y.: Fully polynomial byzantine agreement in t + 1 rounds. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, STOC 1993, New York, NY, USA, pp. 31–41. ACM (1993)
13. Gilbert, S., Kowalski, D.R.: Distributed agreement with optimal communication complexity. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, 17–19 January 2010, pp. 965–977 (2010)
14. Klonowski, M., Kowalski, D.R., Mirek, J.: Ordered and delayed adversaries and how to work against them on a shared channel. Distrib. Comput. 1–25, September 2018
15. Kowalski, D.R., Mirek, J.: On the complexity of fault-tolerant consensus. CoRR, abs/1905.07063 (2019)
16. Kowalski, D.R., Mosteiro, M.A.: Polynomial counting in anonymous dynamic networks with applications to anonymous dynamic algebraic computations. In: 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, 9–13 July 2018, Prague, Czech Republic, pp. 156:1–156:14 (2018)
17. Loui, M.C., Abu-Amara, H.H.: Memory requirements for agreement among unreliable asynchronous processes. Adv. Comput. Res. **4**(163183), 31 (1987)
18. Rabin, M.O.: Randomized byzantine generals. In: 24th Annual Symposium on Foundations of Computer Science (FOCS 1983), pp. 403–409 (1983)
19. Robinson, P., Scheideler, C., Setzer, A.: 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, 6–18 July 2018, pp. 173–182 (2018)