Jagiellonian University
Department of Theoretical Computer Science

Mateusz Pach

Distributed Consensus Protocols

Bachelor's Thesis

Supervisor: dr inż. Krzysztof Turowski

June 2022

Abstract

The distributed consensus is the problem of reaching a consensus between correct processors in the presence of indistinguishable faulty processors that can behave arbitrarily. We implement chosen protocols solving the problem in the Go programming language and compare them later considering different strategies for the faulty processors.

List of Algorithms

Contents

Chapter 1

Introduction

We begin the overview of the problem of the distributed consensus by presenting its history. It originates from the Byzantine generals problem which we are going to describe now.

1.1 The Byzantine generals problem

Reliable distributed systems must be resilient to malfunctions of their components. Such components may send conflicting information across the system. Lamport, Shostak and Pease in [LSP82] proposed a way to avoid conflicts of this type by introducing and solving the Byzantine Generals Problem. The name refers to the following analogy with the Byzantine army: imagine several divisions of the Byzantine army camping outside an enemy city. Every division has its own general. After examining the city each general proposes to attack or to retreat by sending messengers to other generals. Obviously, to maximize the outcome the army should agree on their move. It is not that easy as some of the generals are traitors who can manipulate the others by sending conflicting propositions.

Therefore, the generals need an algorithm which assures that

1. every loyal general decides on the same plan,
2. and small number of traitors cannot make the loyal generals adopt a wrong plan.

The idea for such algorithm is as follows. Let each general obtain the same information about the real propositions of the others. Then, each of them can apply the same rule to make their decision. For example, take the proposition which occurred the most often. The first condition must be met then and the second one as well, unless the numbers of propositions of both types are similar. In such case neither plan can really be wrong. Thus, we now can focus on exchanging the propositions. It would be enough to find a way in which a chosen loyal general's proposition can reach all the loyal generals. In case the general was not loyal we want all the loyal generals to at least agree on one arbitrary proposition from this traitor. We can then apply such a procedure to propagate the propositions of all the generals. The procedure is a solution to the aforementioned Byzantine generals problem.

Let us introduce a formal definition of the Byzantine generals formulated in the terminology of correct and faulty processors, similarly as in [KS08].

Definition 1.1.1 (Byzantine generals problem)**.** We are given n processors. Amongst them, t are faulty and the rest is correct. There is also an input source with an initial value v_s which may be either correct or faulty. The input source broadcasts v_s at the beginning to everyone, unless it's faulty and broadcasts arbitrary values. The processors, being able to communicate, must agree on a certain value v, subject to the following conditions.

- *Agreement* – All correct processors must agree on the same value v.
- *Validity* – If the input source is correct, then v, the agreed upon value by all the correct processors, must be the same as the initial value of the input source v_s.
- *Termination* – Each correct processor must eventually decide on a value v.

1.2 Two other variants

In other papers, we can find two other variants of the Byzantine generals problem. Let us introduce them in a similar way as in [KS08].

Definition 1.2.1 (Interactive consistency problem)**.** We are given n processors. Amongst them, t are faulty and the rest is correct. Each processor i has an initial value v_i, and all the correct processors must agree upon a set of values, with one value for each processor. The formal specification is as follows.

- *Agreement* – All correct processor must agree on the same array of values $A[v_1, \dots, v_n]$.

- *Validity* – If processor i is correct and its initial value is v_i, then all correct processors agree on v_i for $A[i]$. If processor j is faulty, then the correct processors can agree on any value for $A[j]$.
- *Termination* – Each correct processor must eventually decide on the array A.

One can notice it is the same problem we earlier reduced to a few instances of the Byzantine generals problem. The array A stores the agreed information about propositions of generals and v_i is the original proposition of the i-th general.

The second variant is of particular interest for us.

Definition 1.2.2 (Distributed consensus problem)**.** We are given n processors. Amongst them, t are faulty and the rest is correct. Each processor i has an initial value v_i, and all the correct processes must agree upon a single value v, assuring following conditions.

- *Agreement* – All correct processors must agree on the same value v.
- *Validity* – If all the correct processors have the same initial value, then v, the agreed upon value by all the correct processors, must be that same value.
- *Termination* – Each correct processor must eventually decide on a value v.

This problem is very similar to the Byzantine generals problem. You can see the equivalence between them by treating the values received from the input source as the initial values v_i. Agreement and termination conditions are then the same. So is validity if we notice that the correct input source corresponds to equal initial values.

1.3 Theoretical model assumptions

There are several details we have to agree upon before discussing protocols for solving the distributed consensus. In this work we introduce the following notation and assumptions.

Identifiers Each processor has its unique identifier. It is an integer value between 1 and the total number of the processors.

Failures Each faulty processor can behave arbitrarily. It can, but it does not have to, send messages to some or all other processors. In particular, it can stop doing anything. Note that this behavior can sometimes expose the processor's faultiness. It is why, when designing faulty behavior strategies, we make sure that the faulty processor sends messages that it could legally send as a correct processor.

Full connectivity Each pair of processors is connected directly by reliable channels (including self-links). Every processor knows the identifier of the processor on the other end of each channel connected to it. It is achieved by storing the array of channels, in which the i-th entry is the channel connected to the processor with the id i.

Sender authentication It is always known which processor sent a given message.

Synchronous model The processors are embedded in the synchronized model [Lyn96]. Most of the time they do so in order to send messages in rounds. We can then force messages sent in a round to be received in the same round. The synchronization takes place when some processor tries to receive a message. Before it happens, the system waits a reasonable amount of time until all other processors report readiness to receive a message and then all the processors receive the messages simultaneously.

Large processing power of processors The processors are fast enough to assume that local computation happens in a negligible amount of time and communication time is a real bottleneck. They are also sufficiently fast to break any cryptography system based on a time-consuming problem.

Boolean agreement value The initial values are binary for simplification. If we wanted to solve the problem for bigger values, we could encode them in binary and run a few instances of a protocol in parallel.

1.4 Notation

Let us introduce the notation that will be consistently used throughout this paper. Just like in Definition 1.2.2, by n we denote the total number of processors and by t the number of the faulty processors among them. Every processor has its id stored as i. As a reminder, an id is a number between 1 and n which is unique for each processor. Moreover, every faulty processor has its *faulty id* stored as j. The faulty id is also unique for each processor, but it is a number between 1 and t. Each proposed protocol repeats a certain procedure and a single invocation of such procedure will be referred to as *a phase*. During the phase, processors, usually more than once, synchronize to exchange their messages. Such an event is called *a message exchange round*. In pseudocodes of the protocols, one can find $\textsc{Broadcast}(x)$ and $\textsc{Receive}()$ functions. Unless stated otherwise, they work as follows. $\textsc{Broadcast}(x)$ is responsible for sending value x to all processors including the processor from which it is executed. $\textsc{Receive}()$ is used to receive an array of all messages sent in the current message exchange round to the processor executing the function. Messages in the array are ordered by the senders' ids. There is also a $\textsc{Send}(x, i)$ function sending value x to the processor with the id i. Additionally, when considering the faulty behavior strategies, we use $\textsc{PeekV}()$ function which returns an array of all processors' V values. If some processor does not define V, we treat it as *nil* value. Values in the array are ordered by the ids of the processors they come from.

1.5 Distributed Framework

To simulate chosen protocols in a practical environment we implement them in the Go Programming Language [DK15]. The implementations use Distributed Framework [Tur+22] and are incorporated into its repository. We are going to describe how the framework works before heading into details of our implementations.

The crucial part of the framework is a library that enables its users to simulate distributed protocols, which are usually run on multiple processors, on a single processor by using Go language-specific features like simple in use and efficient goroutines and channels. Any user of the framework, besides adding his protocol, needs to only care about setting up a relevant network and a type of communication. Let us list the most important elements of the library.

Nodes Each simulated processor is represented by an object of type `Node`. In particular, nodes provide `SendMessage()` and `ReceiveMessage()` methods used for communication. Moreover, they allow to conveniently share relevant data between rounds and with external observer using `SetState()` and `GetState()` methods.

Graph The nodes can be connected in multiple ways with the use of included graph building functions family. In our code, we are going to use `BuildCompleteGraphWithLoops()` which accordingly to its name constructs and returns a list of nodes connected by channels in a complete graph with loops. It also returns an instance of `Synchronizer` corresponding to the created graph.

Synchronizer Type `Synchronizer` provides a mechanism of synchronization. Its `Synchronize()` method initialize a process. Then, at the beginning of each round nodes need to call `StartProcessing()` method and `FinishProcessing()` method at the end of it. The synchronizer assures that no two nodes are in different rounds, by suspending nodes with `StartProcessing()` until every node calls `FinishProcessing()` in the previous round (if only such exists). The synchronizer also counts the number of rounds and messages sent within all the rounds. One can obtain these statistics using `GetStats()` method.

Besides the library, the framework contains numerous implementations of protocols. These are mainly for the leader election problem on different networks or for some graph problems (e.g. independent set, minimum-weight spanning tree). It also includes the Ben-Or's protocol implementation. As mentioned earlier, in this work we contribute to the Distributed Framework project by opening several accepted pull requests in which we complete items as follows [Pac22a; Pac22b; Pac22c; Pac22d].

1. Organizing and incorporating a unified naming convention for the files in the repository.

2. Adding the single-bit message protocol.

3. Adding the phase king protocol.

4. Refactoring the existing Ben-Or's protocol and unifying it with other distributed consensus protocols.

5. Adding the faulty behavior strategies for all the implemented distributed consensus protocols.

6. Adding the text of current thesis with the script for reproducing the presented figures.

Chapter 2

Protocols

2.1 Overview

Many protocols for solving the distributed consensus problem have been proposed by now. To compare them we use the following parameters.

Resiliency Measures how many faulty processors a given protocol allows, by specifying a minimal number of all the processors in total if there are t faulty ones among them. Its optimal value is proved to be $3t + 1$ in [BGP89]. The same authors show it is possible to practically reach t resiliency, but it requires the use of cryptography.

Number of message exchange rounds Denotes how many times processors need to synchronize and exchange their messages. Fischer and Lynch proved in [FL82] that at least $t + 1$ message exchange rounds are needed to always reach the agreement in a deterministic way.

Maximal message size By this parameter we measure what is the maximal size (number of bits) of a message sent in each round. Its optimum is 1, as the processors need to exchange information.

Local computation Denotes the complexity of the local computation in every processor running a given protocol.

Synchronous vs. Asynchronous model We have promised to consider the synchronous model. There are however protocols working in a harder, asynchronous one, in which the processors cannot synchronize. Interestingly, the protocols in the asynchronous model cannot assure termination and most they can assure is a termination with probability

1 [FLP85]. One can always run such protocols in the synchronous model and even use the fact of running in the easier model to simplify the code. That is why we do not exclude protocols originally designed for the asynchronous model from our considerations.

Let us include a table summarizing the history of deterministic protocols solving the distributed consensus problem in the synchronous model composed by Garay and Moses in [GM98]. Note that the table does not include all the protocols and in some of the referenced papers the problem may be referred to differently e.g. as the Byzantine agreement.

Protocol	resiliency	rounds	messages
[LSP82]	$3t+1$	$t+1$	$\exp(n)$
[Dol+82; TPS85]	$3t+1$	$2t+c$	$\text{poly}(n)$
[Coa86]	$4t+1$	$t+\frac{t}{d}$	$\mathcal{O}(n^d)$
[DRS90; BD91; Coa93]	$\Omega(t^2)$	$t+1$	$\text{poly}(n)$
[Bar+87]	$3t+1$	$t+\frac{t}{d}$	$\mathcal{O}(n^d)$
[MW88]	$6t+1$	$t+1$	$\text{poly}(n)$
[BGP89]	$3t+1$	$t+\frac{t}{d}$	$\mathcal{O}(c^d)$
[BG89]	$4t+1$	$t+1$	$\text{poly}(n)$
[CW92]	$\Omega(t\log t)$	$t+1$	$\text{poly}(n)$
[BG91]	$(3+\epsilon)t$	$t+1$	$\text{poly}(n)\mathcal{O}(2^{\frac{1}{\epsilon}})$
[GM98]	$3t+1$	$t+1$	$\text{poly}(n)$

Table 2.1: History of deterministic protocols for the distributed consensus problem

As we already mentioned, the asynchronous model rejects a possibility of the existence of a deterministic protocol solving the problem. However, randomness can help us. Ben-Or was the first one to make use of it [Ben83]. We do not provide a similar summarizing table for the randomized protocols, as it requires careful examination to avoid overlooking additional assumptions which are commonly added and are not contained in our model e.g. private communication channels or cryptography.

Now, we are going to thoroughly describe, implement and compare variants of protocols proposed in [BG89; BGP89; Ben83].

2.2 Single-bit message protocol

In [BG89] Berman and Garay proposed the single-bit message protocol whose resiliency is $4t + 1$, the number of message exchange rounds is $2t + 2$ and the maximal message length is 1. This is the first protocol in which all these three parameters are simultaneously within a constant factor from their optimal values. It requires polynomial computation time per processor. The protocol is deterministic.

2.2.1 Pseudocode

Algorithm 1 Single-bit message protocol: code for processor i.

```
$V \leftarrow v_i$
for $m \leftarrow 1$ to $t + 1$ do
    BROADCAST($V$)

    $msgs \leftarrow$ RECEIVE()
    $C \leftarrow$ the number of 1's in $msgs$
    if $C \geq n/2$ then
        $V \leftarrow 1$
    else
        $V \leftarrow 0$
        $C \leftarrow n - C$
    end if
    if $m = i$ then
        BROADCAST($V$)
    end if

    $msgs \leftarrow$ RECEIVE()
    if $C < 3n/4$ then
        $V \leftarrow msgs(m)$
    end if
end for
```

2.2.2 Correctness

The protocol runs $t + 1$ phases, each consisting of two message exchange rounds. Each processor has a local variable V which can obtain values from $\{0, 1\}$ and variable C counting messages with value 1 and later with the majority value received from the other processors in the first exchange round. Variables V are set to the initial values at the beginning. In each phase a unique processor becomes a general of the phase (the phase number determines the id of the general). Let us prove the following lemmas.

Lemma 2.2.1. *If all correct processors have the same values V before the phase, then the values are preserved after the phase.*

Proof. If all correct processors have the same values V equal to 1 before the phase, then the condition $C \geq n/2$ is true for all of them since $t < n/4$ and $C \geq 3n/4$, meaning the V values are not changed at line 10. Similarly, if the processors have the same values V equal to 0 the condition is false for all of them and the V values remain not changed at line 8. As C gets adjusted to count the majority value, we have $C \geq 3n/4$, and the V values are preserved as we skip line 19. It proves that values V in correct processors are never changed, hence the claim follows. □

Lemma 2.2.2. *If a general is a correct processor, then after its phase all correct processors have the same values V.*

Proof. If the general is correct, then he broadcast his V at line 14. Two cases may occur now. Either all the correct processors set V to the same value at line 19 or there is some for which $C \geq 3n/4$ at line 18. However $n > 4t$, so the later case implies for some processors there is $C \geq n/2 + t$ at the line. Thus, all the correct processors set V to 1 at line 8 and the value is not changed later, as we observed in the proof of Lemma 2.2.1. Hence, in both cases all the correct processors set V to the same value. □

Lemma 2.2.1 does not only ensure the validity, but also implies it's enough to reach the agreement once, in any phase. At least one of $t + 1$ generals must be correct, so due to Lemma 2.2.2. and the claim above the correct processors must reach an agreement. A finite number of operations guarantees the termination. Thus, we derived a valid protocol.

2.2.3 Optimal faulty behavior strategy

The lack of secure channels assumption in our model can be exploited by the optimal faulty behavior strategy. Such a strategy could possibly get useful information about the correct processors. Because of it, in our considerations for this and next protocols, we assume that the faulty processors have even broader knowledge. Namely, that they can peek at the state of any other processor at any time. In particular they can freely call the PeekV() function.

As we have noticed in Lemma 2.2.1, once the correct processors reach the agreement, the faulty processors cannot ruin it. Thus, the most they can do is not to allow reaching the agreement when one of them is the general of the phase and the agreement is not reached yet. They can do so provided no correct processor has been the general of the phase so far. It turns out they can ensure no agreement only if

$$\mu < 3n/4, \tag{2.1}$$

where $\mu = \max(c_0, c_1)$ and c_i denotes number of correct processors starting with initial value equal to i. Indeed, we observe that otherwise $C \geq 3n/4$ at line 18 in the first phase. It also means that $C \geq n/2$ at line 7 in every correct processor, which implies V is set to 1 in each of them, resulting in the agreement. On the other hand condition in eq. (2.1) is sufficient to construct a working strategy we are looking for.

Algorithm 2 Single-bit message protocol: optimal faulty behavior strategy.

```
 1: for $m \leftarrow 1$ to $t+1$ do
 2:     $vals \leftarrow$ PeekV()
 3:     $E \leftarrow$ the number of 0's in $vals$
 4:     if $j + E < 3n/4$ then
 5:         Broadcast(0)
 6:     else
 7:         Broadcast(1)
 8:     end if
 9:
10:     Receive()
11:     if $m = i$ then
12:         for $k \leftarrow 1$ to $\lfloor n/2 \rfloor$ do
13:             Send($k, 0$)
14:         end for
15:         for $k \leftarrow \lfloor n/2 \rfloor + 1$ to $n$ do
16:             Send($k, 1$)
17:         end for
18:     end if
19:
20:     Receive()
21: end for
```

Let all the aforementioned conditions be met. We will analyze the first phase. There are less than $3n/4$ messages with the value 0 sent in the first message exchange round, because of $\mu < 3n/4$ and the condition at line 4. There are less than $3n/4$ messages with value 1 sent in this round, as $\mu < 3n/4$ and the strategy emits such messages only when it cannot send ones with value 0. If in such case there were at least $3n/4$ messages with value 1, it would mean $(n - t) + t > 2\lfloor 3n/4 \rfloor$ which is impossible. Thus, the correct processors have $C \geq 3n/4$ at line 18. Then, the faulty processor which becomes the general of the phase ensures that less than $3n/4$ correct processors have the same value V after the phase. It implies no agreement is reached by the correct processors after the phase and the agreement. We can also repeat the same reasoning for the next phases by replacing μ with some other value which we know to be less than $3n/4$. Hence, the strategy indeed works.

Like every correct processor, the faulty processor using this strategy sends the messages in the first message exchange round. Also, it sends the messages in the second message exchange if and only if it is the general of the phase. Therefore, the strategy does not trivially expose the faulty processor.

2.2.4 Random faulty behavior strategy

For the sake of future comparisons we also propose a straightforward random faulty behavior strategy.

Algorithm 3 Single-bit message protocol: random faulty behavior strategy.

```
1:  for $m \leftarrow 1$ to $t + 1$ do
2:      for $k \leftarrow 1$ to $n$ do
3:          SEND(0 or 1 with equal probability)
4:      end for
5:
6:      RECEIVE()
7:      if $m = i$ then
8:          for $k \leftarrow 1$ to $n$ do
9:              SEND(0 or 1 with equal probability)
10:         end for
11:     end if
12:
13:     RECEIVE()
14: end for
```

Due to the same reasons as the optimal strategy, this strategy does not trivially expose the faulty processor.

2.2.5 Implementation

There are 5 files directly connected to the protocol.

- `sync_single_bit.go` contains the protocol.
- `sync_single_bit_faulty_behavior.go` contains a faulty behavior factory.
- `sync_single_bit_optimal_strategy.go` contains the optimal faulty behavior strategy.
- `sync_single_bit_random_strategy.go` contains the random faulty behavior strategy.
- `example/consensus_sync_single_bit.go` contains an example of usage.

To run the protocol one has to begin with constructing a complete graph with loops and defining which nodes should run a chosen faulty behavior strategy. Then, we can call the `Run()` function which starts a simulation. Once it is called several grountines are run, one for each node. Instead of looping over phases, each node loops over message exchange rounds. Each iteration comprises synchronization and running a relevant fragment of the protocol. Specifically, the fragment corresponds to one of three parts of pseudocode in algorithm 1 labelled as follows.

- `ER0` – code at line 3.
- `ER1` – code from line 5 to line 14.
- `ER2` – code from line 17 to line 3.

The framework's library requires that every node v sends a message to nodes which call `receiveMessage(`v`)`. To simulate no message the node can send a *nil* message, which is not counted in statistics.

2.3 Phase king protocol

In [BGP89] Berman, Garay and Perry proposed the phase king protocol. Its resiliency is $3t + 1$, the number of message exchange rounds is $3t + 3$ and the maximal message length is 2. It requires polynomial computation per processor. The protocol is deterministic.

2.3.1 Pseudocode

Algorithm 4 Phase king protocol: code for processor i.

```
V ← $v_i$
for $m \leftarrow 1$ to $t+1$ do
    Broadcast($V$)

    $msgs \leftarrow$ Receive()
    $V \leftarrow 2$
    for $k \leftarrow 0$ to $1$ do
        $C(k) \leftarrow$ the number of $k$'s in $msgs$
        if $C(k) \geq n - t$ then
            $V \leftarrow k$
        end if
    end for
    Broadcast($V$)

    $msgs \leftarrow$ Receive()
    for $k \leftarrow 2$ downto $0$ do
        $D(k) \leftarrow$ the number of $k$'s in $msgs$
        if $D(k) > t$ then
            $V \leftarrow k$
        end if
    end for
    if $m = i$ then
        Broadcast($V$)
    end if

    $msgs \leftarrow$ Receive()
    if $V = 2$ or $D(V) < n - t$ then
        $V \leftarrow \min(1, msgs(m))$
    end if
end for
```

2.3.2 Correctness

The protocol runs $t + 1$ phases, each consisting of three message exchange rounds. Each processor has a local variable V which can obtain values from $\{0, 1, 2\}$ and two arrays C and D counting messages with different values received from the other processors in the first and the second message exchange round respectively. Variables V are set to the initial values at the beginning. In each phase a unique processor becomes a king of the phase (the phase number determines the id of the king). Let us now prove the following lemmas.

Lemma 2.3.1. *If all correct processors have the same values V equal to 0 or 1 before the phase, then the values are preserved after the phase.*

Proof. If all correct processors have the same values V equal to $v \in \{0, 1\}$ before the phase, then they all broadcast v at the beginning and for each of them $C(v) \geq n-t$ holds meaning they all set V to v at line 10. For each $k \neq v$ we know that $C(k) < n - t$, as otherwise $n \geq (n - t) + (n - t) - t = 2n - 3t$, but we know it's impossible because $n > 3t$. So V is not set to any other value than v at line 10. It means the processors have the same values V equal to v before the second message exchange round and they all broadcast v at line 13. For each of them $D(k) > t$ holds only for $k = v$, so their values V remain the same before the third message exchange round. The third message exchange round does not alter values V, because $D(v) \geq n - t$ just like $C(v) \geq n - t$. It holds that V is never altered in the correct processors, which concludes the proof. □

Lemma 2.3.2. *If a king is a correct processor, then after its round all correct processors have the same values V equal to 0 or 1.*

Proof. The correct processors' V variables are set to values $\{0, 2\}$ or $\{1, 2\}$ at line 10. Otherwise, if there was a pair of correct processors which set $V = 1$ and $V = 2$ at line 10, then $n \geq 2(n - t) - t$, but $n \geq 3t + 1$, which would mean $3t \geq n \geq 3t + 1$, which is a contradiction. If the king is correct, then he broadcast his V at line 23. Two cases may occur now. Either all the correct processors set V to the same value from $\{0, 1\}$ at line 28 or there is some for which $V \neq 2$ and $D(V) \geq n - t$ at line 27. The later case implies that for each of the correct processor having $V \neq 2$ at line 27 follows $D(V) > t$ (because there are t faulty processors and $n \geq 3t + 1$), meaning that the correct processors have V already set to the same value from $\{0, 1\}$ at line 19, which completes the proof. □

Lemma 2.3.1 does not only assure the validity, but also implies it's enough to reach the agreement once, in any phase. At least one of $t + 1$ kings must be correct, so due to Lemma 2.3.2. and the previous sentence the correct processors must reach an agreement. A finite number of operations assures the termination. Thus, we derive a valid protocol.

2.3.3 Optimal faulty behavior strategy

Similarly as in the previous protocol, we have noticed in Lemma 2.3.1 that in the phase king protocol, once the correct processors reach the agreement, the faulty processors cannot ruin it. Thus again, the best they can do is not to allow reaching the agreement when one of them is the king of the phase and the agreement is not reached yet. They can do so provided no correct processor has been the king of the phase so far. This time, however, there are no additional requirements needed to construct a faulty behavior strategy, which delays the agreement.

Algorithm 5 Phase king protocol: optimal faulty behavior strategy.

```
 1: for $m \leftarrow 1$ to $t+1$ do
 2:     $vals \leftarrow$ PEEKV()
 3:     $E \leftarrow$ the number of 0's in $vals$
 4:     if $j + E < n - t$ then
 5:         BROADCAST(0)
 6:     else
 7:         BROADCAST(1)
 8:     end if
 9:
10:     RECEIVE()
11:     BROADCAST(2)
12:
13:     RECEIVE()
14:     if $m = i$ then
15:         for $k \leftarrow 1$ to $t+1$ do
16:             SEND(0)
17:         end for
18:         for $k \leftarrow t+2$ to $n$ do
19:             SEND(1)
20:         end for
21:     end if
22:
23:     RECEIVE()
24: end for
```

There are less than $n - t$ messages with the value 0 sent in the first message exchange round, because of the condition at line 4. There are less than $n-t$ messages with value 1 sent in this round, as the strategy emits such messages only when it cannot send ones with value 0. If in such case there were at least $n-t$ messages with value 1, it would mean $(n-t)+t > 2(n-t)$, which is impossible. Thus, the correct processors do not reach line 10. It means that they all set V to 2 at line 19. Then, the faulty processor which becomes the king of the phase assures that no agreement is reached by the correct processors after the phase. Thus, the constructed strategy indeed delays the agreement provided that only faulty processors have been the kings of the phase until the current phase.

Like every correct processor, the faulty processor using this strategy sends the messages with a value 0 or 1 in the first message exchange round. Moreover, it always sends the messages in the second message exchange round. Also, it sends the messages in the third message exchange if and only if it is the king of the phase. Therefore, the strategy does not trivially expose the faulty processor.

2.3.4 Random faulty behavior strategy

For the sake of future comparisons we also propose a straightforward random faulty behavior strategy.

Algorithm 6 Phase king protocol: random faulty behavior strategy.

```
1: for $m \leftarrow 1$ to $t+1$ do
2:     for $k \leftarrow 1$ to $n$ do
3:         SEND(0 or 1 with equal probability)
4:     end for
5:
6:     RECEIVE()
7:     for $k \leftarrow 1$ to $n$ do
8:         SEND(0 or 1 or 2 with equal probability)
9:     end for
10:
11:    RECEIVE()
12:    if $m = i$ then
13:        for $k \leftarrow 1$ to $n$ do
14:            SEND(0 or 1 with equal probability)
15:        end for
16:    end if
17:
18:    RECEIVE()
19: end for
```

Due to the same reasons as the optimal strategy, this strategy does not trivially expose the faulty processor.

2.3.5 Implementation

There are 5 files directly connected to the protocol.

- `sync_phase_king.go` contains the protocol.
- `sync_phase_king_faulty_behavior.go` contains a faulty behavior factory.
- `sync_phase_king_optimal_strategy.go` contains the optimal faulty behavior strategy.
- `sync_phase_king_random_strategy.go` contains the random faulty behavior strategy.
- `example/consensus_sync_phase_king.go` contains an example of usage.

The implementation is very similar to the one of the previous protocol. The main difference is that there are four, instead of three, fragments corresponding to the parts of pseudocode in algorithm 4 labelled as follows.

- `ER0` – code at line 3.
- `ER1` – code from line 5 to line 13.
- `ER2` – code from line 15 to line 23.
- `ER3` – code from line 26 to line 3.

2.4 Ben-Or's protocol

In [Ben83] Ben-Or proposed another protocol. Its resiliency is $5t + 1$, the expected number of message exchange rounds is exponential and the maximal message length is 1. It requires polynomial computation per processor. The protocol is probabilistic and it allows it to reach consensus in a constant expected number of message exchange rounds provided we allow $\Omega(t^2)$ resiliency. We present a simplified version of it, as the original one worked in the asynchronous model.

2.4.1 Pseudocode

In this section we modify how Broadcast(x) works in a way that now the processor calling it does not send a message to itself.

Algorithm 7 Ben-Or's protocol: code for processor i.

```
V ← $v_i$
while $True$ do
    Broadcast($V$)

    $msgs$ ← Receive()
    for $k$ ← 0 to 1 do
        $C(k)$ ← the number of $k$'s in $msgs$
    end for
    if $C(0) > (n+t)/2$ then
        Broadcast(0)
    else if $C(1) > (n+t)/2$ then
        Broadcast(1)
    end if

    $msgs$ ← Receive()
    for $k$ ← 0 to 1 do
        $D(k)$ ← the number of $k$'s in $msgs$
    end for
    if $D(0) \geq t+1$ or $D(1) \geq t+1$ then
        if $D(0) \geq t+1$ then
            $V \leftarrow 0$
        end if
        if $D(1) \geq t+1$ then
            $V \leftarrow 1$
        end if
        if $D(0) + D(1) > (n+t)/2$ then
            ignore future alternations of $V$ value
            finish after next phase
        end if
    else
        $V$ ← 0 or 1 with equal probability
    end if
end while
```

2.4.2 Correctness

The protocol runs in phases, each consisting of two message exchange rounds. Each processor has a local variable V which can obtain values from $\{0, 1\}$ and two arrays C and D counting messages with different values received from the other processors in the first and the second message exchange round respectively. Variables V are set to the initial values at the beginning. We proceed to the proofs of the following lemmas.

Lemma 2.4.1. *If all correct processors have the same values V before the phase, then the values are preserved after the phase and it's their second last phase.*

Proof. All correct processors broadcast the same value of V equal to v at the beginning. It means $C(v)$ is set to at least $n-t$ at line 7. But $n-t \geq (n-t)/2$, so everyone broadcasts the same v again at line 10 or line 12. Then, $D(v)$ is set to at least $n-t$ and $D(v')$ is set to at most t at line 17 for any $v' \neq v$. It means the value of V is not altered at lines 21 and 24. We assumed that $n > 5t$, so $n - t \geq (n + t)/2$, which means line 27 is executed. It implies V is never changed during this and future phases. Line 28 assures it is the second last phase. □

Lemma 2.4.2. *If for any correct processor $D(v) \geq t+1$ for some $v \in \{0, 1\}$, then for none of the correct processors $D(\neg v) \geq t + 1$ after line 17.*

Proof. Assume, to the contrary, that it's not the case. It means that at least one correct processor sends 0 and at least one correct processor sends 1 in the second message exchange round. Thus, there's a correct processor executing line 10 and a correct processor executing line 12. Therefore, there are more than $n-t$ messages from the correct processors in the first message exchange round, which is a contradiction. □

Lemma 2.4.3. *If some correct processor reaches lines 27 and 28, then the rest of correct processors reach these lines in the same or next phase. Once all of the correct processors reach these lines, all of them have the same value V.*

Proof. If there's such a processor, then it receives more than $(n + t)/2$ messages in the second message exchange round and $(n - t)/2$ of them are from the correct processors. Let's assume $V = v$ in that processor when it reaches lines 27 and 28. Due to Lemma 2.4.2 the processor receives at most t messages with v', such that $v' \neq v$, in the second message exchange round.

Thus, the number of messages v from the correct processors in that round is at least $(n-t)/2-t$. And $(n-t)/2-t \geq t+1$, as $n > 5t$. It implies all the correct processors set V to v at lines 21 and 24. Also, they don't set V to v' since Lemma 2.4.2 holds. That means all the correct processors reaching lines 27 and 28 in this phase have $V = v$. Moreover, all of the correct processors start the next phase with the same value and applying Lemma 2.4.1 we deduce they preserve that value and reach lines 27 and 28. □

Lemma 2.4.1 does not only guarantee the validity, but also implies it is enough to reach the agreement once, in any phase. Lemma 2.4.3 implies the agreement is reached once all the correct processors terminate. From Lemma 2.4.2, it follows that the probability of all the correct processors having the same values V at the beginning of any phase (excluding the first one) is at least 2^{t-n}. Thus, with probability equal to 1, after an exponential expected number of phases, all the correct processors must terminate. Hence, we proved that the protocol is correct.

2.4.3 Optimal faulty behavior strategy

By Lemma 2.4.1 and Lemma 2.4.3 we know that the faulty processors can have an impact only if the correct processors are not in the agreement before the given phase and no correct processor has reached lines 27 and 28. Moreover, they can prevent the agreement in a given phase only if

$$\mu \leq (n+t)/2, \tag{2.2}$$

where $\mu = max(c_0, c_1)$ and c_i denotes number of correct processors starting with value of V equal to i in this phase. Indeed, if eq. (2.2) holds, then at least $(n+t)/2$ correct processors broadcast the same value in the second message exchange round. In such case, $D(k) \geq t+1$ at line 17 for some k, as $(n+t)/2 > t+1$ when $n > 5t$ and $t \geq 1$. Moreover, $D(0)+D(1) \geq (n+t)/2$ holds at line 17. Therefore, an agreement is obtained and lines 27 and 28 are reached.

Assume these necessary conditions are met. Obviously, the goal of the faulty behavior strategy is to not let any correct processor reach lines 27 and 28. If the strategy succeeds, then at the beginning of the next phase, either all the correct processors have new random values of V or some of them have values of V assigned at line 21 or line 24.

Due to Lemma 2.4.2 all the processors with values assigned at lines 21 and 24 have the same value. Thus, the strategy should force all the correct processors to get new random values of V before the next phase, as otherwise we increase the probability of not satisfying the condition at eq. (2.2) in the next phase. If we manage to find such a strategy, then we can treat the phases independently. This strategy should minimize the number of messages sent in the second message exchange round by both (a) not sending the messages from the faulty processors and (b) reducing the messages sent from the correct processors. This reduction is maximal when, in the first message exchange round, the faulty processors send messages 0 and 1 in such a way that they help violate the conditions at lines 9 and 11 in the correct processors. Ideally, the faulty processors should not be sending any messages in the first message exchange round, but they all have to send them to avoid exposure. Thus, we obtain the following strategy.

Algorithm 8 Ben-Or's protocol: optimal faulty behavior strategy.

```
 1: while $True$ do
 2:     $vals \leftarrow$ PeekV()
 3:     $E \leftarrow$ the number of 0's in $vals$
 4:     if $j + E \leq (n+t)/2$ then
 5:         Broadcast(0)
 6:     else
 7:         Broadcast(1)
 8:     end if
 9:
10:     Receive()
11:
12:     Receive()
13: end while
```

Lemma 2.4.4. *The strategy succeeds to delay the agreement if the condition at eq. (2.2) is met.*

Proof. If the condition is met then the strategy restrains the correct processors from sending any messages in the second message exchange round. It means they cannot reach lines 27 and 28. □

Like every correct processor, the faulty processor using this strategy always sends the messages in the first message exchange round. Therefore, the strategy does not trivially expose the faulty processor.

Condition at eq. (2.2) and Lemma 2.4.4 imply the strategy does not always prevent the agreement, but the more faulty processors there are, the higher probability is they manage to prevent the agreement. Due to earlier observations, it is also the best strategy that can be found.

2.4.4 Random faulty behavior strategy

For the sake of future comparisons we also propose a straightforward random faulty behavior strategy.

Algorithm 9 Ben-Or's protocol: random faulty behavior strategy.

```
while $True$ do
    for $k \leftarrow 1$ to $n$ do
        if $k \neq i$ then
            Send(0 or 1 with equal probability)
        end if
    end for

    Receive()
    for $k \leftarrow 1$ to $n$ do
        $x \leftarrow$ 0 or 1 or 2 with equal probability
        if $k \neq i$ and $x \neq 2$ then
            Send($x$)
        end if
    end for

    Receive()
end while
```

Due to the same reasons as the optimal strategy, this strategy does not trivially expose the faulty processor.

2.4.5 Implementation

There are 5 files directly connected to the protocol.

- `sync_ben_or.go` contains the protocol.
- `sync_ben_or_faulty_behavior.go` contains a faulty behavior factory.
- `sync_ben_or_optimal_strategy.go` contains the optimal faulty behavior strategy.
- `sync_ben_or_random_strategy.go` contains the random faulty behavior strategy.
- `example/consensus_sync_ben_or.go` contains an example of usage.

The implementation is again similar to the ones of the previous protocols. There are three fragments corresponding to the parts of pseudocode in algorithm 7 labelled as follows.

- `ER0` – code at line 3.
- `ER1` – code from line 5 to line 12.
- `ER2` – code from line 15 to line 3.

These are not the only ones, however. As mentioned earlier the framework's library requires that every node v sends a message to nodes that call `receiveMessage(`v`)`. This causes a problem, since the correct processors do not finish at the same phase. We solve this issue by exploiting Lemma 2.4.3. Every correct processor reaching lines 27 and 28, instead of running `ER1` and `ER2` fragments next, runs `ER1Done` and `ER2Done` fragments. These are modifications of the former ones which ignore received messages, do not require any messages sent to them (including *nil* messages) and do not change the value of V. It is possible thanks to `IgnoreFutureMessages()` method which starts an additional goroutine for every node to receive any messages from channels addressed to the given node in an infinite loop. After executing `ER2Done` fragment, the processor finishes.

Chapter 3

Comparison

We want to measure how quickly correct processors reach the agreement in various conditions. We could do so by comparing how many phases each protocol has to run. However, it would be interesting only for the Ben-Or's protocol, as the other two always run exactly for $t + 1$ phases. The fact they always require such number of phases does not mean they do not reach the agreement earlier. Indeed, they do so very often, for example when any processor with an id lower than $t + 1$ happens to be a correct processor. Potentially, there are some applications, in which we do not care about the time needed for the correct processors to be able to assure us that they solved the distributed consensus problem, but only about the time they really used to solve the problem. It is why, instead of counting all the phases performed by a given protocol in some run, we count only the phases before reaching the agreement. In practice, we achieve that by analyzing logged values of V from the correct processors throughout the whole run.

In the comparisons we use notation as follows.

n – the number of the processors.

t – the number of the faulty processors.

b – the percentage of the correct processors initialized with 0 within all the correct processors.

p – the number of the phases before reaching the agreement.

To make the comparisons we perform runs of the protocols in several setups. In all of them $n = 40$. Because results may depend on randomness, each run of the Ben-Or's protocol is repeated 5 times and the average of the results is taken. Similarly, every run with a random faulty behavior strategy is repeated 5 times and the average of the results is taken. It means that every run of the Ben-Or's protocol with a random faulty behavior strategy is repeated 25 times. Whenever we mention b or *all* b, we mean one or all of the $b \in [0, 0.1, 0.2, 0.3, 0.4, 0.5]$, respectively. Note that due to the symmetry of the correct processors it is enough to test any permutation of the initial values for a given b.

3.1 Optimal vs. random faulty behavior strategy

We have devoted a significant part of this work to finding the optimal faulty behavior strategies. A natural idea is to check their effectiveness compared to any other strategies. We compare them with the earlier described random strategies, which are one of the potential strategies one can effortlessly come up with.

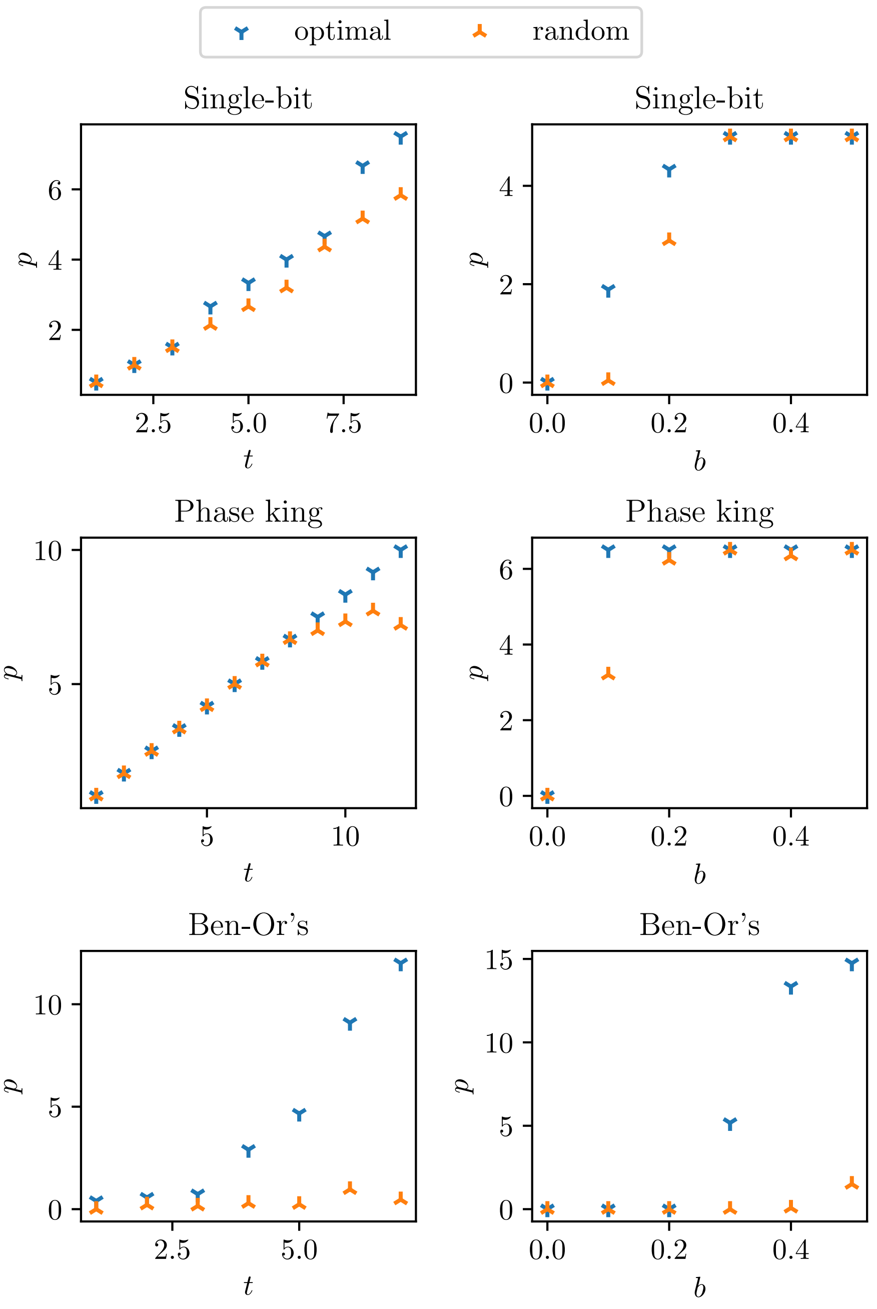
3.1.1 Setups

We aim to show when the difference between the efficiencies of optimal and random faulty behaviour strategies is the largest, so in every setup the faulty processors get the lowest ids. For each of the protocols, runs are performed in the same setups using both strategies. Firstly, for a given legal t the protocol is run with all b and the average of the results is taken. It is how we obtain the left-hand side plots in fig. 3.1. Secondly, for a given b the protocol is run with all legal t and the average of the results is taken. This way we obtain the right-hand side plots in fig. 3.1.

3.1.2 Results

As expected, for all the setups the optimal strategy is always similar to or better than the random one. The more faulty processors there is the bigger difference in effectiveness is observed. It is especially a case in the Ben-Or's protocol where p grows exponentially with respect to t in the setup with the optimal strategy and does not seem to grow in the setup with the random strategy. Moreover, p is highly dependent on b in all the protocols. The closer b is to 0.5 the more effective the faulty behavior strategy is. The optimal faulty behavior strategies outperform the random ones the most for $b = 0.15, b = 0.1, b = 0.4$ in the single-bit message, phase king and Ben-Or's protocols, respectively.

Figure 3.1: Strategies vs. number of faulty processors (left) and percentage of zero-initialized correct processors (right)



3.2 Protocols vs. random faulty behavior strategy

We may also want to consider which of the described protocols is the most effective under similar conditions. In this section, we examine how the protocols cope with random faulty behaviors which can model the case of ”naturally” malfunctioning processors rather than ones intentionally designed to cause the longest delay in reaching the agreement.

3.2.1 Setups

For each of the protocols, runs are performed in the same setups using the random strategy. Firstly, for a given legal t the protocol is run with all b and the average of the results is taken. It is how we obtain the left-hand side plot in fig. 3.2. Secondly, for a given b the protocol is run with all legal t and the average of the results is taken. This way we obtain the right-hand side plot in fig. 3.2. In the runs the faulty processors have random ids. Analogous runs are performed with the lowest ids assigned to the faulty processors and results are presented in fig. 3.3.

3.2.2 Results

Faulty processors with random ids

All of the protocols are observed to reach the agreement within the first two phases with respect to various b and t. Greater b and t result in a bit greater p, especially in the case of the Ben-Or's protocol. The single-bit message and phase king protocols are a bit more resilient to the faulty behavior compared to the Ben-Or's protocol.

Faulty processors with the lowest ids

The phase king and single-bit message protocols reach the agreement within the number of phases proportional to t. However, the former needs more phases. The Ben-Or's protocol needs only 1 phase for all t. The phase king protocol runs, for almost every b, result in the same constant p. The single-bit message protocol runs give similar results, but even smaller p for $b < 0.3$. The Ben-Or's protocol reaches the agreement within the first phase for all b except $b \approx 0.5$. The single-bit message and phase king protocols are less resilient to the faulty behavior compared to the Ben-Or's protocol.

Figure 3.2: Protocols vs. number of faulty processors (left) and percentage of zero-initialized correct processors (right). Faulty processors have random ids

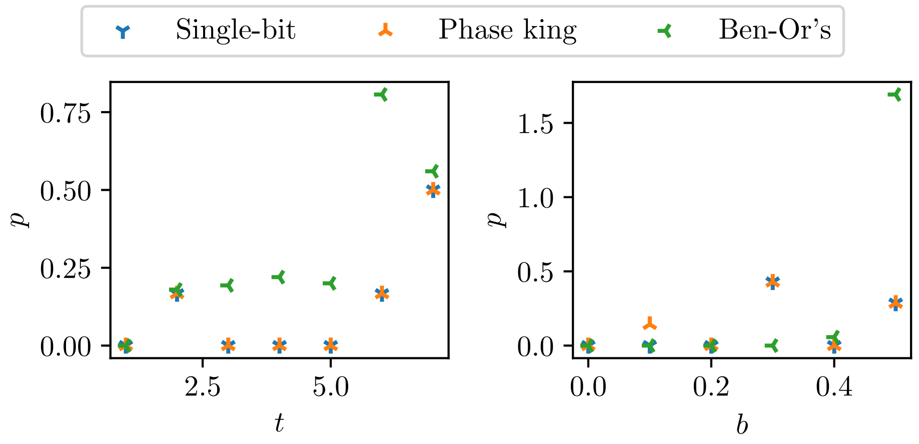
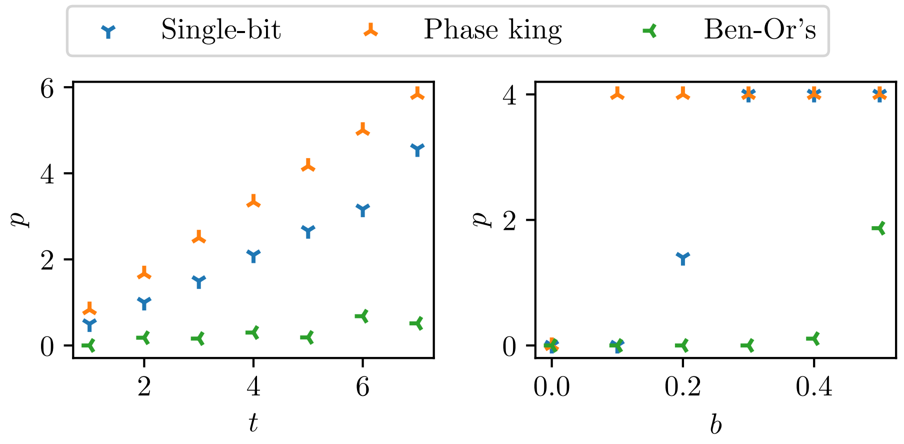


Figure 3.3: Protocols vs. number of faulty processors (left) and percentage of zero-initialized correct processors (right). Faulty processors have the lowest ids



3.3 Protocols vs. optimal faulty behavior strategy

Finally, we consider which of the described protocols is the most effective when facing the optimal faulty behavior strategy. It can model the case of the faulty processors being intentionally designed to cause the longest delay in reaching the agreement.

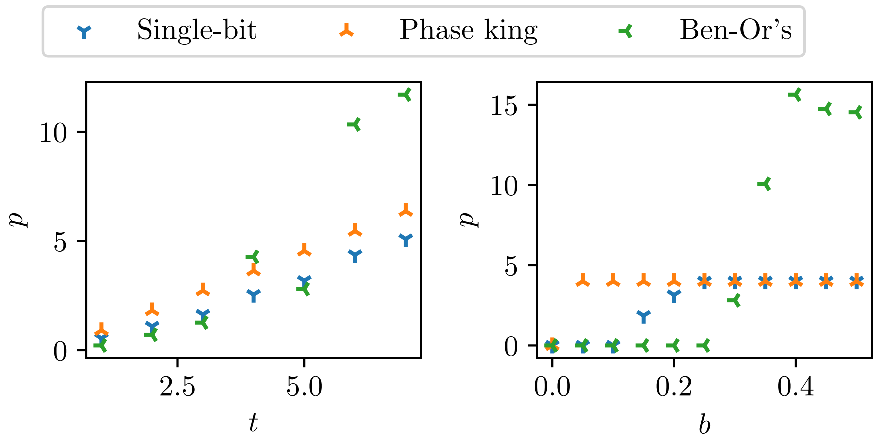
3.3.1 Setups

For each of the protocols, runs are performed in the same setups using the respective optimal strategy. Firstly, for a given legal t the protocol is run with all b and the average of the results is taken. It is how we obtain the left-hand side plot in fig. 3.4. Secondly, for a given b the protocol is run with all legal t and the average of the results is taken. This way we obtain the right-hand side plot in fig. 3.4. In the runs, the faulty processors have the lowest ids, as it either increases or does not affect the effectiveness of the faulty strategies.

3.3.2 Results

All of the protocols are observed to reach the agreement within almost the same number of phases, proportional to t, except for the Ben-Or's protocol which needs twice more phases when run with a large t. The single-bit message and phase king protocols runs, for almost every b, result in the same constant p. The former one results in even smaller p for $b < 0.25$. On the other hand, the Ben-Or's protocol reaches the agreement within the first phase for $b \leq 0.25$, but the p grows exponentially for greater b until $b \approx 0.4$, after which the p is almost constant.

Figure 3.4: Protocols vs. number of faulty processors (left) and percentage of zero-initialized correct processors (right). Faulty processors have random ids



Bibliography

[Bar+87] Amotz Bar-Noy et al. “Shifting Gears: Changing Algorithms on the Fly to Expedite Byzantine Agreement”. In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 1987, pp. 42–51. DOI: 10.1145/41840.41844.

[BD91] Amotz Bar-Noy and Danny Dolev. “Consensus Algorithms with One-Bit Messages”. In: *Distributed Computing* 4 (1991), pp. 105–110. DOI: 10.1007/BF01798957.

[Ben83] 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*. Association for Computing Machinery, 1983, pp. 27–30. DOI: 10.1145/800221.806707.

[BG89] Piotr Berman and Juan Garay. “Asymptotically Optimal Distributed Consensus (Extended Abstract)”. In: *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1989, pp. 80–94. DOI: 10.5555/646243.681443.

[BG91] Piotr Berman and Juan Garay. “Efficient Distributed Consensus with $n = (3 + \epsilon)t$ Processors (Extended Abstract)”. In: *Distributed Algorithms - 5th International Workshop Proceedings.* 1991, pp. 129–142. DOI: 10.1007/BFb0022442.

[BGP89] Piotr Berman, Juan Garay, and Kenneth Perry. “Towards optimal distributed consensus”. In: *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. 1989, pp. 410–415. DOI: 10.1109/SFCS.1989.63511.

[Coa86] Brian Coan. "A Communication-Efficient Canonical Form for Fault-Tolerant Distributed Protocols". In: *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 1986, pp. 63–72. DOI: 10.1145/10590.10596.

[Coa93] Brian Coan. "Efficient Agreement Using Fault Diagnosis". In: *Distributed Computing* 7 (1993), pp. 87–98. DOI: 10.1007/BF02280838.

[CW92] Brian Coan and Jennifer Welch. "Modular Construction of a Byzantine Agreement Protocol with Optimal Message Bit Complexity". In: *Information & Computation* 97 (1992), pp. 61–85. DOI: 10.1016/0890-5401(92)90004-Y.

[DK15] Alan Donovan and Brian Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 2015.

[Dol+82] Danny Dolev et al. "An efficient algorithm for byzantine agreement without authentication". In: *Information and Control* 52.3 (1982), pp. 257–274. DOI: 10.1016/S0019-9958(82)90776-8.

[DRS90] Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. "Early Stopping in Byzantine Agreement". In: *Journal of the ACM* 37 (1990), pp. 720–741. DOI: 10.1145/96559.96565.

[FL82] Michael Fischer and Nancy Lynch. "A Lower Bound for the Time to Assure Interactive Consistency". In: *Information Processing Letters* 14.4 (1982), pp. 183–186. DOI: 10.1016/0020-0190(82)90033-3.

[FLP85] Michael Fischer, Nancy Lynch, and Michael Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Journal of the ACM* 32.2 (1985), pp. 374–382. DOI: 10.1145/3149.214121.

[GM98] Juan Garay and Yoram Moses. "Fully Polynomial Byzantine Agreement for $n > 3t$ Processors in $t + 1$ Rounds". In: *SIAM Journal on Computing* 27.1 (1998), pp. 247–290. DOI: 10.1137/S0097539794265232.

[KS08] Ajay Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[MW88] Yoram Moses and Orli Waarts. “Coordinated traversal: $(t + 1)$-round Byzantine agreement in polynomial time”. In: *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*. 1988, pp. 246–255. DOI: 10.1109/SFCS.1988.21941.

[Pac22a] Mateusz Pach. *Add Distributed Consensus Protocols thesis*. 2022. URL: https://github.com/krzysztof-turowski/distributed-framework/pull/22 (visited on 06/2022).

[Pac22b] Mateusz Pach. *Add optimal faulty behaviors for consensus protocols*. 2022. URL: https://github.com/krzysztof-turowski/distributed-framework/pull/21 (visited on 06/2022).

[Pac22c] Mateusz Pach. *Add phase king protocol*. 2022. URL: https://github.com/krzysztof-turowski/distributed-framework/pull/13 (visited on 06/2022).

[Pac22d] Mateusz Pach. *Add single-bit message protocol*. 2022. URL: https://github.com/krzysztof-turowski/distributed-framework/pull/20 (visited on 06/2022).

[TPS85] Sam Toueg, Kenneth Perry, and T. K. Srikanth. “Fast Distributed Agreement (Preliminary Version)”. In: *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 1985, pp. 87–101. DOI: 10.1145/323596.323604.

[Tur+22] Krzysztof Turowski et al. *Distributed Framework*. 2022. URL: https://github.com/krzysztof-turowski/distributed-framework (visited on 06/2022).