Hybrid Fault-Tolerant Consensus in Asynchronous and Wireless Embedded Systems

Wenbo Xu
Technische Universität Braunschweig, Braunschweig, Germany
wxu@ibr.cs.tu-bs.de

Signe Rüsch
Technische Universität Braunschweig, Braunschweig, Germany
ruesch@ibr.cs.tu-bs.de

Bijun Li
Technische Universität Braunschweig, Braunschweig, Germany
bli@ibr.cs.tu-bs.de

Rüdiger Kapitza[1]
Technische Universität Braunschweig, Braunschweig, Germany
kapitza@ibr.cs.tu-bs.de

Abstract

Byzantine fault-tolerant (BFT) consensus in an asynchronous system can only tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ faulty processes in a group of n processes. This is quite a strict limit in certain application scenarios, for example a group consisting of only 3 processes. In order to break through this limit, we can leverage a hybrid fault model, in which a subset of the system is enhanced and cannot be arbitrarily faulty except for crashing. Based on this model, we propose a randomized binary consensus algorithm that executes in complete asynchrony, rather than in partial synchrony required by deterministic algorithms. It can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ Byzantine faulty processes as long as the trusted subsystem in each process is not compromised, and terminates with a probability of one. The algorithm is resilient against a strong adversary, i. e. the adversary is able to inspect the state of the whole system, manipulate the delay of every message and process, and then adjust its faulty behaviour during execution.

From a practical point of view, the algorithm is lightweight and has little dependency on lower level protocols or communication primitives. We evaluate the algorithm and the results show that it performs promisingly in a testbed consisting of up to 10 embedded devices connected via an ad hoc wireless network.

2012 ACM Subject Classification Software and its engineering → Software fault tolerance

Keywords and phrases Distributed system, consensus, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.15

1 Introduction

Fault-tolerant consensus is one of the fundamental problems in distributed systems. It is an essential component to build more sophisticated distributed applications. Especially with the rapid growth of wireless embedded devices in recent years, systems tend to work cooperatively to achieve a common goal [16, 11, 14]. This requires novel consensus algorithms to adapt to

[1] This work is part of the DFG Research Unit Controlling Concurrent Change, funding no. FOR 1800 and grant no. KA 3171/5-1.

© Wenbo Xu, Signe Rüsch, Bijun Li, and Rüdiger Kapitza;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 15; pp. 15:1–15:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the corresponding system features and application requirements that are inherently different from the quite well-explored wired settings. Firstly, network connectivity is fragile and an upper bound of communication delay may not be guaranteed. Thus an asynchronous system model must be considered. Secondly, the system could be running in an open environment in contrast to more or less static and well maintained environments such as data centers. Furthermore processes might be exposed to extreme temperature, radiation, physical damage and malicious attacks. All these factors lead to a higher demand for fault-resilience. In short, the system has to tolerate as many faulty processes as possible – and not only crashes, but also Byzantine faults. Thirdly, because of the limited resources, the consensus algorithm should have low complexity and less assumptions about low-level network protocols.

In this work we present a novel binary consensus algorithm, Trusted Ben-Or, which is tailored for embedded wireless systems. It is fault-tolerant and can work in a completely asynchronous system. Due to the fact that no deterministic consensus algorithm exists in an asynchronous system with faulty processes [13], Trusted Ben-Or uses randomization to overcome this impossibility. In the end it ensures that all correct processes can terminate with the probability of 1.

Moreover, in an asynchronous system a total of n processes can only tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine processes in order to achieve consensus [8]. This is quite a strict constraint, and it is impossible for a group of only 3 processes to reach BFT consensus. So another important goal of our work is to increase the number of tolerable faulty processes utilizing a slightly extended system model. Many modern processors, including the ones used in the context of embedded systems [3], provide nowadays trusted execution environments, which can be utilized to build a trusted subsystem protected by hardware. The trusted subsystem is tamper-proof and cannot be compromised by a Byzantine host, except for crashing. This technique can be leveraged to prevent equivocation, i. e. sending contradictory messages to different recipients. As a result, the tolerable processes can be increased to $\lfloor \frac{n-1}{2} \rfloor$. The idea that a subsystem is more trustworthy than the remainder of the system can be categorized as a hybrid fault model [21].

In summary, the contributions of this paper include:

- We propose an asynchronous hybrid fault-tolerant consensus algorithm, Trusted Ben-Or, which can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ faulty processes among n processes.
- The algorithm is resilient against a strong adversary model, which means that the adversary can inspect the state of every process and message, and can arbitrarily reorder the message delivery of every process. We provide a correctness proof, which is the first complete proof under this system model to our knowledge.
- Trusted Ben-Or is tailored for wireless embedded systems for its simplicity and low complexity. Every message is sent via broadcast to make full use of the transmission medium. The communication does not require encryption, nor complex communication primitives such as reliable broadcast, nor TCP-like protocols that could be unavailable in certain application domains. Because of the trusted subsystem, the message authentication can use symmetric encryption, which is more efficient than asymmetric digital signatures.
- We implement Trusted Ben-Or and evaluate it in a real wireless ad hoc environment instead of in a pure simulation. The results are promising and comparable to Turquois [19], another well-known wireless ad hoc BFT consensus algorithm.
- We discuss some common issues regarding the termination of randomized BFT consensus algorithms, and point out that some algorithms might not be able to terminate in a strong adversary model.

The rest of the paper is organized as follows: Section 2 defines the system model and the problem statement. Section 3 explains the Trusted Ben-Or algorithm in details. The

correctness proof is given in Section 4. Section 5 discusses the optimization against omission failures and common issues of randomized BFT consensus algorithms. Section 6 shows the evaluation results. Section 7 discusses the related work, and Section 8 concludes the paper.

2 System Model and Problem Definition

In this section we give the system model and the correctness criteria of the consensus problem.

2.1 Processes and Asynchronous System

The system consists of n processes numbering from 1 to n. It is an asynchronous system in which the delay of message transmission or process execution is unbounded. We firstly consider a reliable communication so that every message transmitted between correct processes (defined below) is eventually delivered. Later we will discuss the message omission issue.

For the purpose of presentation, we assume a *global wall-clock*, which is nevertheless not available to the processes. We also model the algorithm execution as a discrete event simulation and assume a virtual *scheduler*. At each time (clock tick) t, the scheduler chooses a process to take a step, e. g. to deliver a message, or to execute a line of the algorithm, etc. We call each step an *event*.

2.2 Strong Adversary Model and Trusted Subsystem

There exists an adversary working against the system. At any time t, the adversary can compromise the current scheduled process to behave abnormally. Such faulty behaviours include: skip some execution steps or stop working permanently (crash), update state or send messages not according to the algorithm, send messages to only a subset of recipients, etc. A process is marked as Byzantine at the time when he performs the faulty behaviours, and will never be regarded as correct afterwards. All the processes not marked as Byzantine are called *correct* at time t. In the remaining of this paper, the statements like "a correct process does something" indicate that the process is still correct at the time when he does the thing. The adversary can compromise at most $f \leq \lfloor \frac{n-1}{2} \rfloor$ processes in total. For simplicity we assume $f = \lfloor \frac{n-1}{2} \rfloor$ in the rest of the paper.

The adversary is a *strong adversary*, meaning that he knows all the previous events, and is able to inspect the current state of every process and the content of every message in the network at any time. He also has the full control over the scheduler to schedule the events arbitrarily. In this way, the adversary can adapt his behaviour to the current system state, leading to a maximum attack power.

However, each process is equipped with a trusted subsystem that cannot be compromised by the adversary and will always operate honestly, unless it is crashed. Furthermore, the adversary does not know the secret key(s) stored in the subsystem, neither can he break the cryptographic mechanisms. As a result, the adversary cannot bypass the trusted subsystem to forge any messages authenticated by the subsystem. As we will show later, the trusted subsystem is used to implement a hybrid fault model [21] similar to *identical Byzantine* [4].

► Remark. A strong adversary seems impractical in real-life scenarios. However, for a distributed algorithm, it is impossible to predict the execution order and message delivery order because of race conditions. There could be corner cases that will hinder the algorithm. In fact, a strong adversary model indicates that each of these corner cases is already tackled. Otherwise, people need to explain why these cases cannot or can hardly happen.

2.3 Problem Definition

Every process can propose a binary value 0 or 1. (A faulty process can propose both values, or no value at all. Any abnormal value other than 0 and 1 will be ignored by correct processes.) The randomized consensus algorithm is called correct, if it fulfils the following properties:

- *Agreement:* if two correct processes decide, they decide the same value.
- *Validity:* if a correct process decides v, $v \in \{0,1\}$ and is proposed by at least $\lfloor \frac{n}{4} \rfloor + 1$ processes.
- *Termination:* any process decides with the probability of 1, or he becomes Byzantine.

Note that the validity implies that the decision can be proposed merely by Byzantine processes. This is different from the *strong validity* in most $3f+1$ consensus problems, which requires that if all correct processes propose v, they must also decide v. However, strong validity is impossible in an asynchronous system with $n < 3f+1$, so the requirement must be relaxed in our case. Yet it is adequate to certain applications, in which both values are acceptable and the processes only need to agree on a common one. Or some extra mechanisms can be applied to detect and eliminate the faulty value (e. g. [10]). Since these can be application-dependent and do not belong to the consensus, we omit the detailed discussion here.

3 Trusted Ben-Or Algorithm

TRUSTED BEN-OR is a randomized consensus algorithm consisting of several asynchronous rounds and each round has two phases, as shown in Algorithm 1. In the first phase, every process broadcasts a *P-message* to propose its current value, together with a flag indicating whether this value is taken deterministically (D-GET) or from a coin flip (R-GET). Upon receiving $\lceil \frac{n+1}{2} \rceil$ P-messages (including his own), a process enters the second phase by broadcasting a *V-message* to vote. He votes for $v \in \{0,1\}$ if all received P-messages have the same value v, otherwise he votes for a default value $\perp$. The process then waits for $\lceil \frac{n+1}{2} \rceil$ V-messages. If all of them vote for the same value $v \in \{0,1\}$, the process can decide on v. If at least one V-message votes for $v \in \{0,1\}$ while the others vote for $\perp$, the process updates his own value to v and sets the flag to D-GET. Otherwise, i. e. all received V-messages vote for $\perp$, the process flips a coin to get a random number, and sets the flag to R-GET.

There is an initialization round (line 4-9), in which each process broadcasts his initial proposal and picks the majority from the received quorum to start round 1.

3.1 Message Authentication and Trusted Coin in the Subsystem

One critically harmful Byzantine fault is equivocation, i. e. sending inconsistent messages to different recipients in a broadcast. Equivocation can be prevented by reliable broadcast [7], which takes several communication rounds and requires $n > 3f$. It can also be avoided by simply using a strict monotonic counter in the trusted subsystem of every process. Every message must be authenticated together with a counter value, and each value can only be used once. More technical details can be referred to works such as [17].

The algorithm also requires a random bit (line 15), so an unbiased trusted coin is placed inside the trusted subsystem. This prevents a Byzantine process from arbitrarily manipulating the result of the coin. Moreover, the message authentication and random number generation should be integrated as an atomic operation. Otherwise, a Byzantine process can repeatedly toss the trusted coin until it obtains the desired result.

Algorithm 1 Trusted Ben-Or algorithm.

```
1  v_D ← v_p                                                    /* Store last D-get value */
2  φ ← 0                                                        /* phase number */
3  flag ← D-GET
4  broadcast ⟨INIT, φ, p, v_D⟩
5  wait for ⌈(n+1)/2⌉ valid ⟨INIT, φ, i, v_i, flag_i⟩ messages from different processes
6  if among them exist ≥ ⌈(n+1)/2⌉/2 messages with value 0
7      v_D ← 0
8  else
9      v_D ← 1
10 φ ← 1
11 loop forever:
12     if flag = D-GET                                          /* P-phase */
13         broadcast ⟨PR, φ, p, v_D, D-GET⟩ with certificate (see Section 3.2)
14     else
15         broadcast ⟨PR, φ, p, coin(), R-GET⟩ with certificate
16     wait for ⌈(n+1)/2⌉ valid ⟨PR, φ, i, v_i, flag_i⟩ messages from different processes
17     if all messages of carry the same v:                     /* V-phase */
18         broadcast ⟨VO, φ, i, v⟩ with certificate
19     else:
20         broadcast ⟨VO, φ, i, ⊥⟩ with certificate
21     wait for ⌈(n+1)/2⌉ valid ⟨VO, φ, i, ∗⟩ messages from different processes
22     if all messages of above line have the same value v ≠ ⊥:
23         decide v
24     if received at least one ⟨VO, φ, i, v_i⟩ with v_i ≠ ⊥:   /* Update */
25         v_D ← v_i
26         flag ← D-GET
27     else:
28         flag ← R-GET
29     φ ← φ + 1
```

The trusted subsystem maintains a unique identifier uid, a monotonically increasing counter value u, and secrete key(s) to calculate message authentication codes. The keys cannot be disclosed to the non-trusted part of the system. It provides the following APIs:

- `authenticate`(m, u): Takes a message m and a counter value u; Requires u greater than its last accepted value; Outputs an authentication code based on $m||uid||u$.
- `authenticate_with_coin`(m, ptr, u): Takes an extra pointer pointing to a bit of m; Requires u greater than its last accepted value; Firstly fills the bit where ptr points to with the trusted coin toss result, then outputs an authentication code based on $m||uid||u$.
- `verify`(m, uid, u, AC): Checks whether the authentication code AC is generated based on $m||uid||u$ by the trusted subsystem uid.

Now we define the corresponding counter value u that is uniquely bound to every message. More specifically, $INIT$ message must be authenticated with counter value $u = 0$. Every P-message of round ϕ must have $u = [\phi|0]$, where "|" is the separation of the least significant bit and higher bits. And every V-message must have $u = [\phi|1]$. Note that at line 15 of Algorithm 1, $\langle PR, \phi, p, \texttt{coin}(), \textit{R-GET}\rangle$ is authenticated by invoking `authenticate_with_coin`, and the result of the a coin toss is filled into the place where `coin()` stands.

3.2 Message Certificate and Validation

Every message needs to be proved that it is congruent with the algorithm specification. To achieve this, a process is required to provide a set of previously received messages, named

certificate, when he broadcasts a new message. The certificate is piggybacked with the newly sent message. A message is called *valid* only if it includes the correct certificate. The certificate of every message is defined as follows:

1. $\langle INIT, 1, *, v, D\text{-}GET\rangle$ is valid if $v \in \{0, 1\}$ without any certificate.
2. $\langle PR, 1, *, v, D\text{-}GET\rangle$ requires $(\lfloor \frac{n}{4} \rfloor + 1)\langle INIT, 0, *, v\rangle$ for $v = 0$, or $(\lfloor \frac{n+2}{4} \rfloor + 1)$ $\langle INIT, 0, *, v\rangle$ for $v = 1$.[2]
3. $\langle PR, \phi, *, v, D\text{-}GET\rangle(\phi > 1)$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$.
4. $\langle PR, \phi, *, v, R\text{-}GET\rangle(\phi > 1)$ requires $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp\rangle$.
5. $\langle VO, \phi, *, v\rangle$ with $v \in \{0, 1\}$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, *\rangle$. Furthermore, if there is one $\langle PR, \phi, *, v, D\text{-}GET\rangle$ in the certificate, the certificate of this message, i. e. $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$, must also be included.
6. $\langle VO, 1, *, \perp\rangle$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, 1, *, *, D\text{-}GET\rangle$ with both 0 and 1 are proposed.
7. $\langle VO, \phi, *, \perp\rangle(\phi > 1)$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, *\rangle$ with both 0 and 1 are proposed, plus $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp\rangle$.

The certificates are carefully designed to ensure the correctness of the algorithm. There are two important properties. Firstly, a process can immediately validate a message based on the certificate, and does not rely on previously received messages. Secondly, the messages in the certificate do not require further certificate for themselves. Otherwise the message size will grow infinitely. However, this gives a Byzantine process the chance to bypass the validity check, because he can include some other invalid messages into the certificate, turning a faulty message into valid. But this issue does not affect the correctness, as we will see later.

The cases 1, 2, 4 and 6 are trivial as they directly follow the algorithm execution. The other three cases deserve some explanations.

- In case 3, if a correct process i sends a $\langle PR, \phi, i, v, D\text{-}GET\rangle(\phi > 1)$, he must have received at least one valid $\langle VO, \phi - 1, *, v\rangle$ (line 24). It is pointless to use this single message as a certificate, because this message itself can be invalid. But if i is correct, he must have validated the $\langle VO, \phi - 1, *, v\rangle$, which contains $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$ according to case 5. So he must include these $\lceil \frac{n+1}{2} \rceil$ messages into his own certificate.
- In case 5, $\langle VO, \phi, i, v\rangle$ implies that the process i has received $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, *\rangle$ (line 17), which must be included in the certificate. The extra requirement is only necessary for the termination as we will prove later. If i is correct, he must have checked the validity of any $\langle PR, \phi, *, v, D\text{-}GET\rangle$ before putting it into certificate. So he is required to strip the certificate of that message and put into his own certificate.
- In case 7, the process sends $\langle VO, \phi, i, \perp\rangle(\phi > 1)$ because he has received $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, *\rangle$ proposing different values (line 19). As we will see later, it is impossible to see both valid $\langle PR, \phi, *, 0, D\text{-}GET\rangle$ and $\langle PR, \phi, *, 1, D\text{-}GET\rangle$ in the same round $\phi > 1$, so one of the different values must come from a $\langle PR, \phi, *, *, R\text{-}GET\rangle$ caused by a coin flip, whose certificate is $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp\rangle$. So i must strip the certificate from this *R-GET* message and include it into his own certificate.

According to the explanation above, we can conclude the following lemma:

▶ **Lemma 1.** *If a correct process is about to broadcast a message, he is able to attach a corresponding certificate to that message immediately.*

---

[2] Because $\lfloor \frac{n}{4} \rfloor + 1 = \lceil \frac{n+1}{2} \rceil / 2$, i. e. a majority of the quorum. And if a tie between 0 and 1 appears in the quorum, we choose 0 here. This can be modified to 1, depending on the application.

4 Correctness Proof

In this section we prove the correctness of Trusted Ben-Or.

4.1 Agreement

We first show that if any correct process decides any value v, then from the next round, no one can generate a valid P-message to propose another value.

▶ **Lemma 2.** *If a correct process decides v in round ϕ, the only valid P-message of $\phi + k$ is $\langle PR, \phi + k, *, v, D\text{-}GET \rangle$ for any $k > 0$.*

Proof. We start with $k = 1$. A correct process only decides v if there are $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, v \rangle$. This excludes the existence of any valid $\langle PR, \phi+1, *, *, R\text{-}GET \rangle$ that requires $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, \bot \rangle$ as certificate. Because 2 quorums or $\lceil \frac{n+1}{2} \rceil$ processes intersect with at least one process, the equivocation mechanism ensures that a process cannot vote for v and $\bot$ in the same round. And each $\langle VO, \phi, *, v \rangle$ contains $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, * \rangle$ in its certificate. So for the same reason, $\langle PR, \phi+1, *, 1-v, D\text{-}GET \rangle$, which requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, 1-v, * \rangle$, cannot become valid, either. On the other hand, $\langle PR, \phi + 1, *, v, D\text{-}GET \rangle$ can be valid because its certificate exists, making it to be the only valid P-message of $\phi + 1$.

Now assume that the only valid P-message of $\phi + k$ is $\langle PR, \phi + k, *, v, D\text{-}GET \rangle$ for some $k > 0$. Then all correct processes only broadcast $\langle PR, \phi + k, *, v, D\text{-}GET \rangle$. This makes $\langle PR, \phi + k + 1, *, 1 - v, D\text{-}GET \rangle$ invalid. Since all correct processes do not accept any invalid $\langle PR, \phi+k, *, 1-v, * \rangle$, they can only broadcast $\langle VO, \phi+k, *, v \rangle$. So $\langle PR, \phi+k+1, *, *, R\text{-}GET \rangle$ cannot be valid because less than $\lceil \frac{n+1}{2} \rceil$ vote for $\bot$. Thus $\langle PR, \phi + k + 1, *, v, D\text{-}GET \rangle$ is the only valid P-message. Using induction we can confirm this lemma. ◀

The agreement property directly ensues:

▶ **Theorem 3.** *No two correct processes decide differently.*

Proof. We prove it by contradiction. Suppose that two correct processes decide v in ϕ and $1-v$ in ϕ' respectively. Apparently $\phi \neq \phi'$, because $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, v \rangle$ and $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, 1 - v \rangle$ cannot exist at the same time. Assume $\phi < \phi'$, according to Lemma 2, the only valid P-message of ϕ' is $\langle PR, \phi', *, v, D\text{-}GET \rangle$. But correct process decides $1 - v$ in ϕ' means that he has received $\lceil \frac{n+1}{2} \rceil$ valid $\langle VO, \phi', i, 1-v \rangle$ messages certified with $\lceil \frac{n+1}{2} \rceil \langle PR, \phi', *, 1-v, * \rangle$. This leads to a contradiction, because no correct process will propose $1-v$ in a P-message. ◀

4.2 Termination

The proof of termination is similar to the proof of [2], but is more complex due to the Byzantine behaviours. We first show that every correct process is able to start any round, then prove that there is some "lucky" round in which every correct process can decide.

▶ **Lemma 4.** *Every correct process is able to start any round $\phi \geq 1$.*

Proof. It clearly applies for $\phi = 0$ and 1. Now assume that every correct process starts a round $\phi \geq 1$. According to Lemma 1, each correct process is able to assemble a certificate and broadcast a valid P-message. So eventually there are at least $\lceil \frac{n+1}{2} \rceil$ valid P-messages in the system, enabling correct processes to terminate the wait of line 16 and then broadcast a valid V-message. Again there are at least $\lceil \frac{n+1}{2} \rceil$ valid V-messages eventually, so every correct process can terminate the wait of line 21 and start the next round $\phi + 1$. Using induction we confirm that every correct process can start any round $\phi \geq 1$. ◀

▶ **Corollary 5.** *In every round ϕ, at least one of the three P-message forms is valid: $\langle PR, \phi, *, 0, D\text{-}GET\rangle$, $\langle PR, \phi, *, 1, D\text{-}GET\rangle$, $\langle PR, \phi, *, *, R\text{-}GET\rangle$. And at least one of the three V-message forms is valid: $\langle VO, \phi, *, 0\rangle$, $\langle VO, \phi, *, 1\rangle$, $\langle VO, \phi, *, \perp\rangle$*

In order to show a lucky round will eventually happen, we adopt the similar definition of [2], but with some modifications due to the presence of Byzantine faults:

▶ **Definition 6.** A value $v \in \{0, 1\}$ is ***ϕ*-major** at time t_0, if $\geq \lceil \frac{n+1}{2} \rceil$ processes have created the message $\langle PR, \phi, *, v, *\rangle$ and authenticated with the monotonic counter at t_0. A value $v \in \{0, 1\}$ is ***ϕ*-locked** at time t_0, if 1) no valid $\langle PR, \phi, *, 1 - v, *\rangle$ with a certificate exists before t_0 and 2) from t_0 on, no $\langle PR, \phi, *, 1 - v, *\rangle$ can be created, or such a message can never collect a certificate.

In other words, v is ϕ-locked at t_0 means that we are sure that no valid $\langle PR, \phi, *, 1 - v, *\rangle$ can exist at any time. Obviously, if v is ϕ-major or ϕ-locked at t_0, then v is also ϕ-major or ϕ-locked at any time $t \geq t_0$.

▶ **Lemma 7.** *If a value v is ϕ-locked at some time, then every correct process can decide v by the end of round $\phi + 1$.*

Proof. All correct processes will start round ϕ and they only propose v. They will not accept any P-message with value $1 - v$, since it is invalid. So they only vote for v, leading to less than $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, \perp\rangle$, therefore $\langle VO, \phi + 1, *, \perp\rangle$ can never become valid. Neither $\langle VO, \phi + 1, *, 1 - v\rangle$ can be valid, because of the lack of certificate. According to Lemma 4 and Corollary 5, a correct process can complete collecting valid V-messages at line 21, and the only valid form is $\langle VO, \phi + 1, *, v\rangle$. As a result, every correct process can decide v by the end of $\phi + 1$. ◀

In the rest part we will show if the trusted random number generators of every process happen to generate a sequence of lucky results, one value will become locked. We start with the following lemma:

▶ **Lemma 8.** *If v is ϕ-major at t_0, and if the trusted random number generator happens to output v for every process creating $\langle PR, \phi + 1, *, v, R\text{-}GET\rangle$ at any time (can be before t_0), then v is $(\phi + 1)$-locked at t_0.*

Proof. There is no $\langle PR, \phi + 1, *, 1 - v, R\text{-}GET\rangle$ because of the trusted random number generator. And a $\langle PR, \phi+1, *, 1-v, D\text{-}GET\rangle$ cannot be valid any more, because v is already ϕ-major and no certificate containing $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, 1 - v, *\rangle$ can exist. ◀

Starting from $\phi = 2$, we group every 3 rounds into an *epoch*. So the r-th epoch ($r \geq 1$) consists of rounds $3r - 1$, $3r$ and $3r + 1$. And we define two oracle functions. The oracles know the state of the whole system, but are not available to the processes. The `first_toss`(r, t) oracle returns the time $t_a \leq t$, when the first correct process executes line 15 to flip a coin to create $\langle PR, 3r, *, v, R\text{-}GET\rangle$ in round $3r$. If no correct processes ever did that, the oracle returns NaN. Note that the correct process i needs only to be correct until t_a. Also note that the function can return $t_a = t$, i. e. a correct process is executing line 15 exactly at time t.

The `lucky_coin`$(r, \phi, t) \rightarrow \{0, 1\}$ oracle assesses whether a random bit obtained in round ϕ, at time t is lucky or not. The return value of `lucky_coin`(r, ϕ, t) is defined as following:

(i) `lucky_coin`$(r, 3r + 1, t)$ returns 1 for any t;
(ii) `lucky_coin`$(r, 3r - 1, t)$ and `lucky_coin`$(r, 3r, t)$ return 1, if `first_toss`(r, t) returns t_a, and 0 is not $(3r - 1)$-major at time t_a;
(iii) `lucky_coin`$(r, 3r - 1, t)$ and `lucky_coin`$(r, 3r, t)$ return 0 in cases other than (ii).

Obviously, as soon as a process (correct or Byzantine) has executed line 15 to flip the coin, we can immediately know whether the result is lucky or not. The reason is that the return values of both `first_toss`(r, t) and `lucky_coin`(r, ϕ, t) are determined merely by the events happened before or at t, and are independent of any future events.

We say an epoch is *lucky*, if every coin toss at time t of line 15 gets the consistent result of `lucky_coin`(r, ϕ, t). Let t_b be the time when every correct process has completed round $3r + 1$. Such a t_b must exist (Lemma 4), and the system state has only three possibilities:

1. `first_toss`(r, t_b) returns NaN, meaning that no correct process ever tossed a coin;
2. `first_toss`(r, t_b) returns t_a and 0 is $(3r - 1)$-major at time t_a;
3. `first_toss`(r, t_b) returns t_a and 0 is not $(3r - 1)$-major at time t_a.

For the three cases, there are the following lemmas:

▶ **Lemma 9.** *(Case 1) If `first_toss`(r, t_b) returns NaN, then some value v is $(3r+1)$-locked at t_b.*

Proof. All correct processes have completed phase $3r$ and no one created $\langle PR, 3r, *, *, R\text{-}GET\rangle$, so they all must have created $\langle PR, 3r, *, v, D\text{-}GET\rangle$ with the same v.

If $v = 0$: because `lucky_coin`$(r, 3r, t)$ must return 0 due to its definition (case (iii)), so all $\langle PR, 3r, *, v', R\text{-}GET\rangle$ messages (must be created by Byzantine processes) have $v' = 0$. And there are no valid $\langle PR, 3r, *, 1, D\text{-}GET\rangle$ messages, so 0 is $3r$-locked, thus also $(3r+1)$-locked.

If $v = 1$: since all correct processes propose 1 in round $3r$, 1 is $3r$-major at time t_b. According to case (i) of the definition of `lucky_coin`, in round $3r + 1$, the lucky coin always returns 1. So 1 is $(3r + 1)$-locked because of Lemma 8. ◀

▶ **Lemma 10.** *(Case 2) If `first_toss`(r, t_b) returns t_a and 0 is $(3r - 1)$-major at time t_a, then 0 is $3r$-locked at t_b.*

Proof. Because 0 is $(3r - 1)$-major at time t_a, it is also $(3r - 1)$-major at time t_b. For any time $t < t_a$, `lucky_coin`$(r, 3r, t)$ must return 0; for any time $t \geq t_a$, `lucky_coin`$(r, 3r, t)$ must return 0 as well. So every $\langle PR, 3r, *, v, R\text{-}GET\rangle$, whenever it is created, must have $v = 0$. According to Lemma 8, 0 must be $3r$-locked. ◀

▶ **Lemma 11.** *(Case 3) If `first_toss`(r, t_b) returns t_a and 0 is not $(3r - 1)$-major at time t_a, then 1 is $(3r + 1)$-locked at t_b.*

Proof. A correct process creates a $\langle PR, 3r, *, v, R\text{-}GET\rangle$ at time t_a (here $v = 1$). This indicates that he must have received $\lceil \frac{n+1}{2} \rceil \langle VO, 3r - 1, *, \perp\rangle$, among which at least one is from a correct process. That process must have received at least one valid $\langle PR, 3r-1, *, 1, *\rangle$. This P-message cannot have the flag $R\text{-}GET$, because any coin tossed before t_a must get 0 (case (iii) of `lucky_coin`$(r, 3r - 1, t)$). So there is one valid $\langle PR, 3r - 1, *, 1, D\text{-}GET\rangle$, but there is never valid $\langle PR, 3r - 1, *, 0, D\text{-}GET\rangle$. From time t_a on, the lucky coin only returns 1 for phase $3r - 1$ according to case (ii), so there is no more $\langle PR, 3r - 1, *, 0, R\text{-}GET\rangle$ after t_a. Thus the total number of $\langle PR, 3r - 1, *, 0, R\text{-}GET\rangle$ is less than $\lceil \frac{n+1}{2} \rceil$ forever. So we can confirm that there is never a valid $\langle VO, 3r - 1, *, 0\rangle$, because 1) it cannot include an invalid $\langle PR, 3r - 1, *, 0, D\text{-}GET\rangle$ into the certificate (recall the definition of the certificate), and 2) the number of $\langle PR, 3r - 1, *, 0, R\text{-}GET\rangle$ can never reach $\lceil \frac{n+1}{2} \rceil$.

Now that there are no valid $\langle VO, 3r-1, *, 0\rangle$, and the lucky coin only returns 1 for phase $3r$ after t_a, so no correct process will create $\langle PR, 3r, *, 0, *\rangle$. Eventually 1 will be $3r$-major. And because the lucky coin always returns 1 in round $3r + 1$, 1 must be $(3r + 1)$-locked because of Lemma 8. ◀

▶ **Lemma 12.** *If epoch r is lucky, all correct processes can decide no later than round $3r+2$.*

Proof. According to Lemmas 9, 10 and 11, some value v must be $3r$-locked or $(3r+1)$-locked at some time. So all correct processes can decide in round $3r+1$ or $3r+2$ (Lemma 7). ◀

▶ **Theorem 13.** *The probability that all correct processes decide is 1.*

Proof. An epoch is lucky if and only if all results from the random number generator coincide with the definition of `lucky_coin`. Each of these coincidences has the probability of 0.5, and is independent with each other. Every epoch may contain at most $3n$ random numbers, so the probability that an epoch is lucky is at least $(0.5)^{3n}$. So the probability that a lucky epoch eventually occurs is $1-(1-p^{3n})^{\infty}=1$. Lemma 12 ensures that all correct processes must decide immediately after such a lucky epoch. ◀

4.3 Validity

▶ **Theorem 14.** *If a correct process decides v, $v \in \{0,1\}$ and is proposed by at least $\lfloor \frac{n}{4} \rfloor + 1$ processes.*

Proof. All messages are required to carry a value $v \in \{0,1\}$, so correct processes cannot decide other values. Assume v is proposed by less than $\lfloor \frac{n}{4} \rfloor + 1$ processes. Then there is no valid $\langle PR, 1, *, v, * \rangle$ in the first round because of the lack of $INIT$ messages, which means v is 1-locked. Every correct process will decide $1-v$ by the end of round 2 (Lemma 7), so no correct one will decide v. ◀

5 Optimization and Discussion

Now we discuss some common issues in the proof of randomized consensus algorithms, as well as the optimization against omission failures, given that the reliable communication without message omissions is sometimes impractical in certain applications.

5.1 Randomization and Strong Adversary

Like most randomized and round-based algorithms, the termination of Trusted Ben-Or relies on a set of processes to luckily obtain the preferred coin values in certain rounds. The definition of a lucky coin value is not trivial, but we argue that this is necessary in a strong adversary model. As mentioned, the luckiness of a coin only depends on the current system state, but not on any future events. Otherwise, suppose that a coin is only lucky if something in the future happens, the adversary could take actions to prevent this from occurring. If an algorithm design violates this principle, it can hardly withstand a strong adversary.

We have considered several other algorithms (e. g. [10, 19]). Similar to Trusted Ben-Or, by the end of each round every process gets an updated value, either deterministically or randomly, and any two correct processes can only deterministically get the same value (0 or 1) in this round. So they conclude that, if all correct processes deterministically or randomly get the same v, they can decide. However, there is a corner case: if the correct processes start with different values, a strong adversary can hinder the termination indefinitely. Because by manipulating the message delivery order, the adversary can firstly let a correct node randomly obtain a value v, then lets another correct node deterministically update to $1-v$ in the same round. As a result, all the correct processes again have different values at the beginning of a new round.

5.2 Handling Omission Failures

In real-world networks, especially in wireless ad hoc networks, links are not always reliable and messages could get lost. In this case, we can consider a *fair-loss link* model, meaning that if a process p sends a message infinitely many times to q, q will then deliver the message infinitely many times. Trusted Ben-Or cannot directly work under a fair-loss link model, so we modify the algorithm by introducing another two tasks running in parallel to Algorithm 1. Task 1 is to periodically broadcast the last sent message. Task 2 is to let a process "jump" to a future round or phase, if it has received a valid message from that round or phase. More specifically, if a process receives a valid $\langle PR, \phi', *, v, \textit{D-GET}\rangle$ or $\langle VO, \phi', *, v\rangle$ that is more advanced than its current state, it will send a message with the same content, and update its state correspondingly. If a $\langle PR, \phi', *, *, \textit{R-GET}\rangle$ is received, besides updating the state, the process has to toss its own coin to create the $\langle PR, \phi, i, \texttt{coin}(), \textit{R-GET}\rangle$.

With this modification, the agreement of the algorithm still holds, because the correctness of Lemma 2 and Theorem 3 only relies on the equivocation prevention and certificate mechanism. However, the termination becomes problematic. Assume the corner case where a correct process receives all P-messages but misses all V-messages in each round. Then it can never decide. But we can guarantee that no correct process gets blocked at any round:

▶ **Lemma 15.** *At any time t_1, for any correct process in round ϕ_1, there is a time $t_2 > t_1$ when the process is in a new round $\phi_2 > \phi_1$, or it becomes Byzantine.*

Proof. Suppose there is a correct process p that stays forever in round ϕ_1 after time t_1. Then apparently there is no correct process entering any round $\phi_2 > \phi_1$ while also staying correct forever, otherwise it will periodically broadcast messages of round ϕ_2 or later rounds. Eventually p can receive at least one of them and can jump to a new round. So all other correct processes keep staying in rounds $\leq \phi_1$ after time t_1. Because p is infinitely broadcasting its message of ϕ_1, all correct processes will eventually receive it and will enter and stay in ϕ_1. Eventually at least one correct process can receive the messages from all correct processes, and can then enter $\phi_1 + 1$. This leads to a contradiction. ◀

Therefore, we have to assume a more relaxed model than fair-loss link, for instance every message gets an independent probability of omission (similar to [20]). The probability can vary from time to time but cannot always be 0. Then if there is a lucky epoch and all messages in this epoch as well as the following round are delivered, all the correct processes can decide. (They still have to handle the outdated V-messages for decision.) This assumption is close to the real network conditions. As we will see later, during our experiments we have observed regular packet losses (24%), but the algorithm can still terminate with relatively low latency.

6 Evaluation

Trusted Ben-Or is evaluated on a testbed consisting of 10 nodes of Raspberry Pi 3 (model B). Each node has a Quad-Core ARM Cortex-A53 1.2 GHz CPU, 1 GB RAM and a 2.4 GHz 802.11n wireless module. The trusted subsystem is built on top of the Open Portable Trusted Execution Environment (OP-TEE) [18] based on ARM TrustZone [3].[3] The algorithm is implemented in C++, except that the trusted functions (counter authentication and random number generation) are implemented in C because of lacking of C++ support in OP-TEE. For comparison, we implement Turquois on the same system as well.

[3] According to this disclaimer (`https://github.com/OP-TEE/build/blob/master/docs/rpi3.md`), Raspberry Pi does not provide an enough trust level. It is only used for demonstration purpose.

The 10 nodes are distributed in different rooms and the farthest distance is about 20 meters. They are connected with a wireless ad hoc network, and all messages of Trusted Ben-Or are sent via UDP multicast. The minimal, median and maximum of the round trip time of an ICMP ping message is 5.6 ms, 12.5 ms and 1356.7 ms respectively. With the *iperf3* tool we also test the UDP link, and the result between two nodes reports the jitter as 139.9 ms, which stands for a high variance of the communication delay, and 24% packet loss rate. So compared to a simulated network, this testbed can reflect a real network environment more closely. Because of the packet loss, we implemented the optimization of Section 5.2.

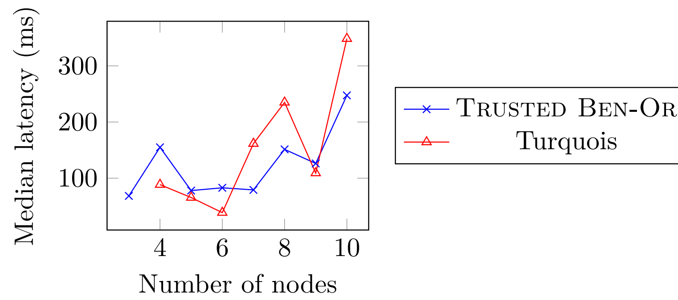
For the experiment, all nodes are connected to a signal machine via Ethernet, and wait for the start signal to start the consensus algorithm almost simultaneously. The nodes are equally divided to be assigned with 0 and 1 as their initial proposals. The performance is evaluated as the delay from the time when nodes start the algorithm until the non-faulty ones decide. For each system setting, we repeat the experiment 100 times.

Firstly the fault-free case is evaluated, and the results are presented in Figure 1. The curve of Turquois starts with 4 because it cannot work with 3 nodes. Figure 1a shows that the median of latency of Trusted Ben-Or has no significant difference compared to Turquois, especially when we take the high jitter value into account. With an increasing of group size, both algorithms tend to take longer time to terminate. The highest median latency of Trusted Ben-Or is 247.4 ms observed in a group with 10 nodes. Besides the median value, we also evaluate the variance of the latency as presented in Figure 1b. It shows that from $n = 8$, the variance increases notably in Trusted Ben-Or. Occasionally (with $\leq 10\%$ probability) it takes 2-3 seconds to terminate. In contrast, the variance of Turquois increases only slowly with n.

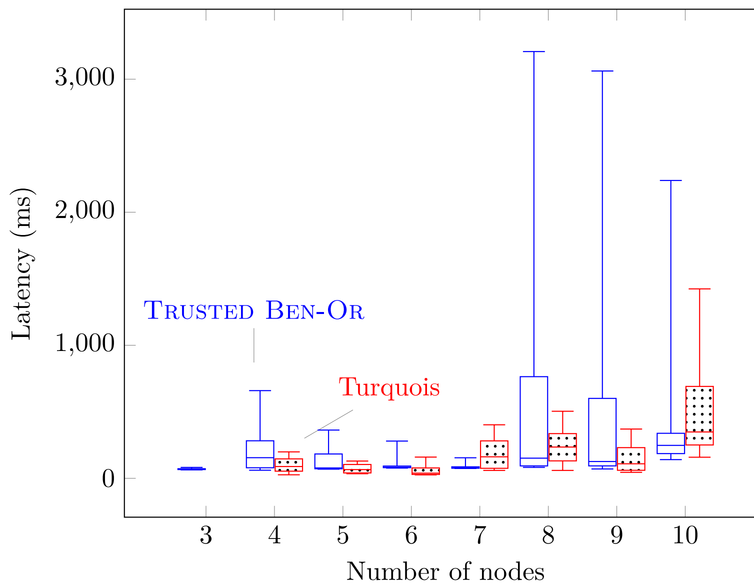
In the second experiment, we inject Byzantine faults into the system to evaluate the fault resilience of Trusted Ben-Or. More specifically, we let $\lfloor \frac{n-1}{2} \rfloor$ nodes act as Byzantine processes. Whenever they are about to send a value of 0 or 1 (if not from the trusted coin), they flip the value to the opposite and then send it; and if they are sending a $\perp$, they do not change it. Furthermore, we let Byzantine nodes not perform the validation at all, so an invalid message could still be included in the certificate of a message voting for $\perp$. As a result, there are fewer valid messages with value 0 or 1, while more valid messages voting for $\perp$. This will hinder the correct nodes achieving consensus. We inject faults into Turquois as well, but only $\lfloor \frac{n-1}{3} \rfloor$ nodes are Byzantine. Since there is no equivocation prevention mechanism in Turquois, we let faulty nodes always send two opposite values if their values are 0 or 1.

We compare the two algorithms with the existence of Byzantine faults and the results are shown in Figure 2. The difference between the median values of latency again is not significant, but both algorithms take longer to terminate. For example with $n = 10$, they need twice as much time as in the fault-free case. Regarding the variance, Turquois is not much affected by the Byzantine faults. But in Trusted Ben-Or the variance becomes higher. In the worst case, the latency could exceed 5 s with $n \geq 8$. This is partly because there are more Byzantine processes in Trusted Ben-Or.

We can conclude that for the most cases, Trusted Ben-Or does not introduce extra overhead when tolerating more faulty processes compared to Turquois. But the variance of Trusted Ben-Or is higher, and more sensitive to the Byzantine faults. Besides the different number of Byzantine processes, another reason is that Turquois has one more phase of message exchange in each round. At a first glance, this is counter-intuitive since one more phase means more communication delay. But in Turquois, in the second phase every process broadcasts its proposal and picks the majority as its vote in the third phase. So if a node is faster in progress, it is more likely to disseminate his proposal to the others because they



(a) Median of latency.



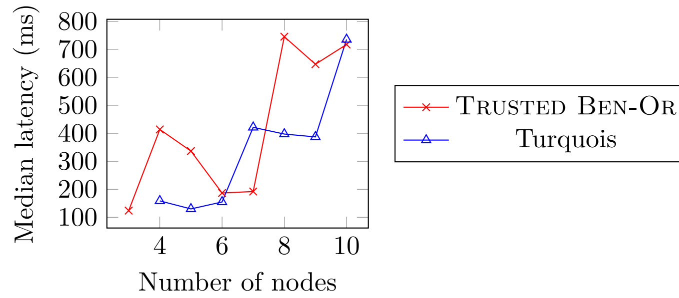
(b) Box plot of latency. The three horizontal bars of the box represent the first, second (median) and third quantiles respectively; the whiskers correspond to the 9th and 91st percentiles respectively.

Figure 1 Latency of consensus in fault-free case.

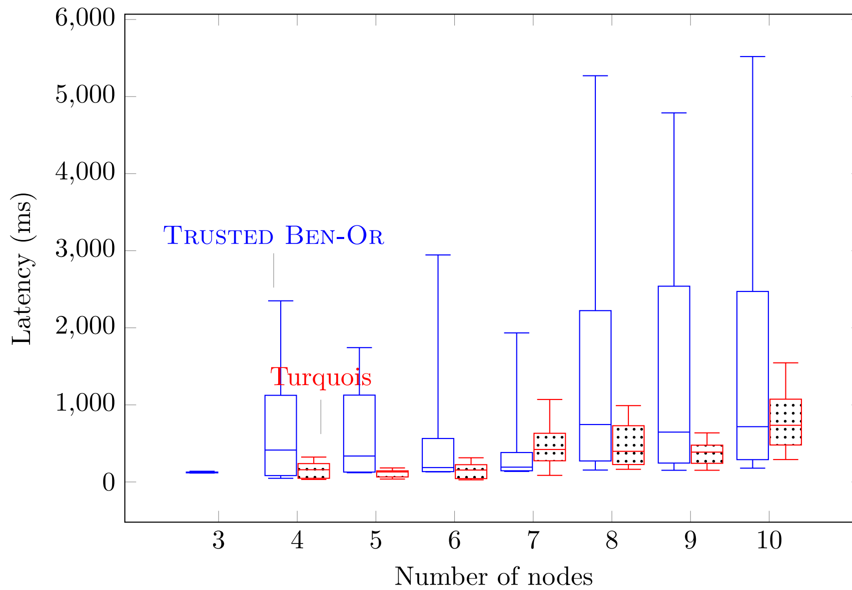
communicate via multicast broadcast. As a result, the nodes are more likely to receive the same vector of proposals, then pick the common value for the third phase. On the other hand, TRUSTED BEN-OR has only two phases in each round. If they propose different values, very likely they have to toss the coin to start the next round. In fact, we have seen more than 50 rounds until they all decide in a group with 10 nodes.

7 Related Work

The impossibility result by Fischer, Lynch and Paterson [13] precludes any deterministic consensus algorithm in asynchronous systems with even a single faulty process. There are two approaches to bypass this impossibility. Firstly we can resort to a weakened system model, e. g. a partially synchronous model [12]. The other option is to use a randomized algorithm (although the correctness criterion is correspondingly weakened from "must terminate" to "terminates with a probability of 1").



(a) Median of latency.



(b) Box plot of latency. The meaning of the boxes is the same as Figure 1b.

Figure 2 Latency of consensus when Byzantine processes exist.

Ben-Or's algorithm [6] is a randomized fault tolerant consensus algorithm for a completely asynchronous system and can withstand a strong adversary. It has a crash fault tolerant variant and a BFT variant. The former can tolerate $f \leq \lfloor \frac{n-1}{2} \rfloor$ crashed processes, which has inspired this work. The BFT version requires $n > 5f$, which is less attractive in practice. Bracha's algorithm [7] improves the maximum tolerable faults to $f \leq \lfloor \frac{n-1}{3} \rfloor$ at the cost of using reliable broadcast, which can introduce considerable overhead. Turquois [19] has a novel message validation mechanism to get rid of the reliable broadcast primitive. Meanwhile, it utilizes an efficient message authentication approach and UDP broadcast, making it tailored for wireless embedded systems. The authors did not mention the strong/weak adversary model, but it turns out that Turquois cannot withstand a strong adversary, as discussed in Section 5.1. Several works have explicitly addressed the weak adversary model [9, 1], in which the adversary does not know everything about the whole system state. They achieve high efficiency at the cost of having this weakened adversary model. Vavala and Neves also propose a speculative randomized consensus algorithm in a so-called *normal condition* where the adversary model is further relaxed compared to the worst case [20]. It is worth mentioning the correctness proof of the crash fault tolerant Ben-Or's algorithm [2]. It gives

a good example of how to proof termination under a strong adversary model. But if we take Byzantine faults into account, the proof becomes more complex as we show in this work.

The hybrid fault model [21] is a common approach to increase the maximum tolerable faulty processes and to decrease the complexity of consensus. In this model, a small subset of the system is trusted and cannot be arbitrarily faulty, but can only fail by crashing. One common usage of this subsystem is to prevent equivocation by using one or more monotonic counters for message authentication [17, 22, 15, 5, 23], so that a Byzantine process cannot send contradictory messages to different recipients — similar to the identical Byzantine fault model [4]. Among them, Ratcheta [23] is also tailored for wireless embedded systems. But all the above mentioned works are deterministic algorithms and assume partial synchrony. Correia et al. [10] discuss the transformation from a crash consensus to Byzantine consensus in asynchronous systems by means of using the hybrid fault model. Although they provide an idea to transform the original Ben-Or's algorithm, above all they require the reliable broadcast primitive. Besides that, the algorithm relies on a failure detector that can eventually find out all Byzantine processes, but the design of this failure detector is unclear. Moreover, they do not mention a trusted random number generator, so the termination seems problematic.

8 Conclusion

In this work, we present Trusted Ben-Or, a randomized hybrid fault-tolerant consensus algorithm. It is operational in an asynchronous system and is resilient against a strong adversary, as long as the adversary cannot compromise the trusted subsystem of each process. Trusted Ben-Or increases the maximum tolerable Byzantine processes to $\lfloor \frac{n-1}{2} \rfloor$. The algorithm is tailored for wireless embedded systems because it does not rely on connection-oriented communication protocols, e. g. TCP, or any complex communication primitives. Neither does it require expensive asymmetric digital signatures. We evaluate Trusted Ben-Or on a testbed consisting of 10 Raspberry Pis connected via an ad hoc wireless network. The results show that the median latency is below 250 ms, and for most of the time Trusted Ben-Or achieves almost the same performance as Turquois – another well-known asynchronous BFT consensus algorithm tolerating only up to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine processes.

References

1. Ittai Abraham, Danny Dolev, and Joseph Y Halpern. An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 405–414. ACM, 2008.
2. Marcos K Aguilera and Sam Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012.
3. ARM. ARM Security Technology - Building a Secure System using TrustZone Technology, ARM Technical White Paper, 2009.
4. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, pages 255–256. Wiley Series on Parallel and Distributed Computing. Wiley, 2004.
5. Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237. ACM, 2017.
6. 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*, pages 27–30. ACM, 1983.

7 Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162. ACM, 1984.

8 Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26. ACM, 1983.

9 Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51. ACM, 1993.

10 Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous Byzantine consensus with 2f+ 1 processes. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 475–480. ACM, 2010.

11 J. Q. Cui, S. K. Phang, K. Z. Y. Ang, F. Wang, X. Dong, Y. Ke, S. Lai, K. Li, X. Li, F. Lin, J. Lin, P. Liu, T. Pang, B. Wang, K. Wang, Z. Yang, and B. M. Chen. Drones for cooperative search and rescue in post-disaster situation. In *2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, pages 167–174, 2015. doi:10.1109/ICCIS.2015.7274615.

12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

13 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

14 Zafar Iqbal, Kiseon Kim, and Heung-No Lee. A cooperative wireless sensor network for indoor industrial monitoring. *IEEE Transactions on Industrial Informatics*, 13(2):482–491, 2017.

15 Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In European Chapter of ACM SIGOPS, editor, *Proceedings of the EuroSys 2012 Conference*, pages 295–308, 2012.

16 Yasuhiro Kuriki and Toru Namerikawa. Consensus-based cooperative formation control with collision avoidance for a multi-UAV system. In *American Control Conference (ACC), 2014*, pages 2077–2082. IEEE, 2014.

17 Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *NSDI*, volume 9, pages 1–14, 2009.

18 Linaro. Open Portable Trusted Execution Environment. https://www.op-tee.org, visited in September, 2018.

19 Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 537–546. IEEE, 2010.

20 Bruno Vavala and Nuno Neves. Robust and speculative Byzantine randomized consensus with constant time complexity in normal conditions. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 161–170. IEEE, 2012.

21 Paulo E Veríssimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006.

22 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *Computers, IEEE Transactions on*, 62(1):16–30, 2013.

23 Wenbo Xu and Rüdiger Kapitza. RATCHETA: Memory-bounded Hybrid Byzantine Consensus for Cooperative Embedded Systems. In *Reliable Distributed Systems (SRDS), 2018 IEEE thirty-seventh Symposium on*. IEEE, 2018.