Do Not Overpay for Fault Tolerance!

Edo Roth   Andreas Haeberlen

University of Pennsylvania

Abstract—In this paper, we argue that distributed real-time and embedded systems sometimes "overpay" for fault tolerance, by using a protocol that is more powerful than what is actually needed, or by failing to take advantage of unique features in these systems. As a result, these systems sometimes perform more computation or communication than is strictly necessary, or they can be unnecessarily complex, and thus more difficult to analyze.

We take a look at the design space for two common problems, broadcast and consensus, and we show that, in a number of scenarios that would be common in real-time systems, these problems have trivial solutions. We then examine two solutions from the literature and propose alternatives that are substantially simpler, less expensive, and more reliable.

I. Introduction

Faults are a fact of life when building distributed systems, and it is important to ensure that the system can tolerate at least some of them. This is particularly true for real-time and embedded systems, which often interact with the physical world. In this setting, a failure at the wrong moment can cause substantial damage, or even loss of life.

The traditional assumption is that fault tolerance should be done using one of several off-the-shelf protocols, such as Paxos [38] (for crash faults) or PBFT [12] (for Byzantine [39] faults). These protocols are general and offer very strong properties, which makes them an attractive choice. However, they also come with a rather substantial cost: in order to tolerate up to f faults, Paxos requires $2f + 1$ replicas, while PBFT requires $3f + 1$; both protocols also have a high message complexity. This hefty "price tag" can be a problem – especially in embedded systems, where computation and bandwidth resources are often scarce.

In this paper, our goal is to show that the cost of robust fault tolerance can often be far lower than is commonly believed. There are two reasons for this. The first is that standard building blocks, such as Paxos and PBFT, were designed for systems with very different properties: for instance, they often assume an asynchronous system with point-to-point links, which complicates the problem substantially and results in worse lower bounds. Thus, when a real-time system with a bus topology and/or tight synchronization uses these protocols, it is effectively "overpaying" for features it does not need! In particular, the strong synchrony, which is a key property of real-time systems but is not usually present in other systems, provides several imporant benefits.

The second reason is that, while non-crash faults such as memory corruption are a common concern for embedded systems, these faults represent only a small subset of Byzantine faults – and it happens to be an "easy" subset! Much of the cost and complexity of Byzantine-tolerant protocols is spent on handling *adversarial* behavior, such as equivocation; this kind of behavior is not likely to happen due to random corruption, and its probability can be reduced further, e.g., by transmitting several copies of messages, or by repeating computations and cross-checking the results. Thus, tolerating "benign" non-crash faults is often much cheaper and easier than tolerating arbitrary Byzantine faults.

We take a look at the design space for fault-tolerant protocols, and we show how the cost can vary substantially, depending on the scenario and the assumptions that are being made. These dependencies can be very subtle, so choosing the protocol that is "best deal" for a given scenario often requires a very close look. We then apply our lessons to two protocols from the literature: we show that the cost of RT-ByzCast [37] can be reduced substantially if it can take advantage of a bus topology, and we show that the consensus protocol by Gujarati et al. [27] can be simplified considerably by taking advantage of the fact that, although the paper describes its fault model as "Byzantine faults" and uses an approach that can tolerate such faults, it does not actually consider adversarial behavior and in fact assumes only benign non-crash faults. In both cases, our proposed replacements are so simple that they are almost trivial, and their timing properties are easy to see, even without a complicated analysis.

The building blocks we use for our solutions are not new – they go back to the early 1980s – and the connection we point out is of the kind that seems immediately obvious once one has become aware of it. However, we do not think that it was obvious to begin with, otherwise the earlier papers would have taken advantage of it. Relative to the current state of the art, as represented by an award paper from last year's RTAS [27], our proposals are much simpler, considerably more reliable, much easier to analyze, and they have storage and communication overheads that are orders of magnitude lower.

II. When is fault tolerance trivial?

When building fault tolerance solutions, a critical and very difficult building block is *agreement*, that is, getting all the replicas to take a consistent action. We begin by examining two agreement problems that are very common: broadcast and consensus.

In the following, we will assume a system of N nodes, $n_1, \ldots, n_N$, that are connected by a network. We say that a node n_i is *correct* in a given execution of this system if it faithfully executes the instructions that were assigned to it; otherwise we say that n_i is *faulty*. Although our focus is on

synchronous systems, which can provide timing guarantees, we consider both synchronous and asynchronous systems here.

A. Broadcast

In the *broadcast problem*, some node n_i sends a value, and the other nodes must each deliver the same value to some local application – say, some replicated state machine. This is sometimes explained using the analogy of a commander and several lieutenants [39]: the commander is supposed to issue a single order, and the lieutenants want to make sure that they all take the same action, even if some lieutenants do not receive the order or a malicious commander gives different orders to different lieutenants. A solution must satisfy the following four properties:

- **Agreement:** If a correct node delivers a value v, then all other correct nodes eventually deliver v.
- **Validity:** If n_i is correct and sends value v, all correct nodes must eventually deliver v.
- **Integrity:** Correct nodes deliver at most one value, and any delivered value must have been sent by n_i.
- **Termination:** Correct nodes deliver a value eventually.

This problem has several names (for instance, it is sometimes called "terminating reliable broadcast", to distinguish it from other variants), and the properties are sometimes stated or named slightly differently.

B. Consensus

The *consensus problem* is quite similar, except that now every node n_i proposes its own value v_i, and the goal is for the correct nodes to decide on a *single* value from among the ones that have been proposed. The properties are:

- **Agreement:** If a correct node decides on a value v, then all other correct nodes eventually decide v.
- **Validity:** If all nodes that propose a value propose the *same* value v, then all correct nodes eventually decide v.
- **Integrity:** Correct nodes decide on at most one value, and only on a value that has been proposed by some node.
- **Termination:** Correct nodes decide on a value eventually.

As with agreement, the properties are sometimes stated differently – for instance, validity is sometimes restricted to values proposed by correct nodes.

C. Fault models

The difficulty of the above two problems depends, to a great extent, on two things: 1) what we mean by "faulty", and 2) the exact properties of the system. In this paper, we consider the following three common fault models:

- **Crash faults:** In this model, faulty nodes simply stop sending messages and executing instructions. For instance, a node might suddenly lose power.
- **Probabilistic faults:** In this model, nodes experience random malfunctions with a certain probability. For instance, a node might experience memory corruption.
- **Byzantine faults:** In this model [39], faulty nodes can do almost anything; they can perform arbitrary computations or send arbitrary messages, and they can collude with

each other. For instance, a faulty node might have been hacked and taken over by a malicious adversary.

Notice that some events, such as message corruption or memory corruption, can occur both in the probabilistic and in the Byzantine model; the difference is in the process that produces them. For instance, an adversary can cause the nodes under her control to consistently flip a certain bit in every single message they send. The same corruption pattern could also occur with probabilistic faults, but it is extremely unlikely.

We assume that, in the crash and Byzantine models, there is an upper bound f on the number of nodes that can be faulty; if a majority of the nodes can fail at the same time, it is often impossible to find any solution at all. For the probabilistic model, we simply assume that faults occur independently with some probability; thus, a majority of the nodes *could* fail at the same time, it is just not very likely.

D. System model

In terms of system properties, we consider the following:

- **Synchrony:** In a synchronous system, there are bounds on computation times and message delays, and nodes have access to closely synchronized clocks. Computation can be thought of as a sequence of discrete rounds, with messages sent in round r being received in round $r + 1$. In an asynchronous system, computations and message transmissions can take arbitrarily long, and clocks are not synchronized.
- **Network topology:** Some systems have a broadcast channel (say, a CAN bus, or switched Ethernet) that delivers a transmitted message to all the nodes. Other systems have unicast channels, such as direct point-to-point links between the nodes.
- **Network ordering:** The network can deliver messages in the order in which they were sent, or it can deliver the messages in any order.
- **Network reliability:** The network can be reliable, that is, it can guarantee that every transmitted message is delivered to its recipient, or it can drop and/or corrupt messages with a certain probability.
- **Transferable authentication:** With authentication, nodes can identify the sender of a given message, even if the message has been forwarded by another node. Without authentication, nodes can only identify the sender of messages that are sent directly to them.

Solutions can guarantee the properties from Sections II-A and II-B either *perfectly*, in the sense that the solution must never violate them in any execution, or *probabilistically*, in the sense that the properties must hold with some high probability.

E. Trivial solutions

Both broadcast and consensus have a reputation for being difficult, but this is not necessarily the case. Consider a synchronous system that already has a reliable broadcast channel at the network level (e.g., a bus), and assume that nodes can fail only by crashing. Then Algorithm 1 trivially solves the broadcast problem. In the first round, n_i uses the network-level

Algorithm 1 A trivial broadcast algorithm (T-Bcast)
1: **procedure** PROPOSE(v)
2:     broadcast(v)
3: **procedure** ROUND(r)
4:     **if** v is received **then**
5:         decide(v)
6:     **else**
7:         decide($\perp$)

Algorithm 2 A trivial consensus algorithm (T-Cons)
1: **procedure** PROPOSE(v_i)
2:     broadcast(v_i)
3: **procedure** ROUND(r)
4:     $V = (\perp, \ldots, \perp)$
5:     $V[i] = v_i$
6:     **if** v_j is received from n_j **then**
7:         $V[j] = v_j$
8:     decide($f(V)$)

broadcast primitive to send the value v it wants to propose; because the network is reliable and the system is synchronous, this broadcast is received by each node in the second round, and each node simply decides v. If n_i crashes before it can send the broadcast, none of the other nodes receive a message, and they each simply decide a default value $\perp$.

In this model, consensus is not much more difficult; Algorithm 2 shows a simple solution. In the first round, each node n_i uses the network-level broadcast primitive to send its proposed value v_i; since the network is reliable and the system is synchronous, each node will receive the same set of messages in the second round. The nodes can then assemble the received values into a vector and apply any deterministic function f to choose one of the elements from the vector (other than $\perp$). Since the vectors will be identical on all the nodes, each node thus decides on the same value.

Of course, the two algorithms are simple because we have chosen idealistic assumptions. But this is precisely our point! We argue that, rather than reflexively reaching for Paxos or PBFT, *one should always ask why the above algorithms are not sufficient*. We discuss some possible reasons next.

F. What if the network is unreliable?

One very common objection to the above model is that, in a practical system, the network is not reliable – there is always a small chance that a given message is lost. There are two ways to respond to this concern, depending on whether, in any given run, we need the system to achieve all four consensus properties with certainty (perfect correctness), or only with high probability (probabilistic correctness). The latter is often acceptable, as long as the probability of failure can be bounded and factored into the system-level reliability analysis.

Perfect: If we insist on getting all four properties with certainty, then the problem cannot be solved with an unreliable network *at all* [25], since it is possible that the network might lose every single message that is ever sent. Because of this, one often assumes something slightly stronger – perhaps that messages are received eventually, if they are transmitted often enough [22]. But since it is impossible to predict how often a message might need to be retransmitted, there is no bound on message delay, so we have no choice but to treat the system as asynchronous. But once we give up on timing, we might as well treat the network as reliable: we can imagine each algorithm being augmented with a little loop that periodically retransmits every message that has been sent, until it is either acknowledged or the algorithm terminates.

Probabilistic: If we are willing to accept a small probability of failure, the solution is even easier than the above simple algorithms. Suppose we expect the algorithm to send up to m messages, each message is dropped with probability p_{drop}, and we are willing to tolerate a failure probability p_{fail}. If we send k copies of each message (and have the recipient drop duplicates), the probability that at least one copy will arrive is $1 - p_{drop}^k$, so we can simply choose k such that $(1 - p_{drop}^k)^m \geq 1 - p_{fail}$.

Of course, this brute-force solution is not necessarily the most efficient choice: for instance, one could use erasure coding to reduce the amount of redundant data that needs to be sent.

Summary: If the algorithm must succeed with certainty, a system with an unreliable network must be treated as asynchronous. If a small probability of failure is acceptable, the network can be made "reliable enough" with retransmissions.

G. What if the network can corrupt messages?

Another reason why our trivial solution might not suffice is that messages might be corrupted in the network; for instance, the network might sometimes randomly flip some bits. As above, the response depends on whether we want a perfect solution, or just one that works with very high probability.

Perfect: If we insist on all three properties holding with certainty, the situation is roughly comparable to the one with Byzantine faults on the nodes. To see why, consider an execution e in which some nodes are malicious and send different messages than the ones they were supposed to. Then we can construct an equivalent execution e' with just message corruptions, in which the nodes originally send the correct messages and every bit in a message that is different in e and e' is then flipped by the network. (The probability of this happening by chance is admittedly very small, but it is not zero.) True, there are some differences between the models – for instance, a malicious node might send extra messages, whereas most corruption models assume that only existing messages can be corrupted – but, in practice, we will usually need to go with a Byzantine-tolerant algorithm.

Probabilistic: If we can tolerate a small probability of failure, the situation changes: we can simply include in each message m a cryptographic hash $H(m)$ of the contents, and we can have the nodes drop any incoming messages $\langle m, h \rangle$ where $h \neq H(m)$. In the random oracle model [7], the chances of a corruption leaving a k-bit hash unchanged is 2^{-k}, which is about $6.8 \cdot 10^{-49}$ for a SHA-1 hash – small enough that

it should be safe to ignore in practice. Thus, the problem effectively reduces to that of an unreliable network, which we have discussed above.

Two possible objection to the use of hash functions are that 1) they might be expensive, and 2) the random-oracle assumption might be wrong. The former should rarely be a concern in practice: for instance, SHA-1 can be computed in less than 200 cycles per byte on an 8-bit CPU [48]. The latter is a fair point: the outputs of a hash function are not really random. There is indeed a certain structure, and, once it is properly understood, this structure can allow *an adversary* to find hash collisions *relatively* quickly. For instance, in 2017, the first SHA-1 collision was found, using about 110 GPU years and 6,500 CPU years [56]; this is why SHA-1 is no longer considered acceptable for security uses. However, the chance of a non-malicious network "finding" a collision instantaneously through random bit flips still seems minuscule.

Summary: If probabilistic guarantees are acceptable and one is willing to trust hash functions, message corruptions can be reduced to message drops.

H. What if node memory can be corrupted?

A third reason for rejecting our trivial solution is that nodes might experience random memory corruption and thus might perform computations incorrectly.

Perfect: If we insist on perfect guarantees, this situation does indeed force us to use Byzantine-tolerant algorithms. To see why, consider an execution e in which a malicious adversary installs some evil software S on a node n_i; we can construct an execution e' in which n_i's original software is transformed into S by a – very unlikely – long sequence of memory corruptions.

Probabilistic: Perhaps surprisingly, the situation is again different if we can accept probabilistic guarantees. Although it may at first appear as if memory corruption errors are a perfect example of Byzantine faults, they are in fact only a *subset* of Byzantine faults; the Byzantine model also includes scenarios in which nodes are compromised by an adversary who might cause the nodes to take the worst possible action *every single time*. To see the difference, consider what happens if we ask a node compute the same function $f(x)$ k times. If the node is controlled by an adversary, the adversary can cause the function to return the same incorrect result every time. But if the node is "just" experiencing random memory corruption, the chances of this happening are small: each of the k invocations has a nonzero chance of executing correctly, and even if two executions do experience corruption, they might still return different results.

Thus, there is a way to essentially reduce memory corruption errors to crash faults: we can repeat computations a few times, and crash the node when the results do not match. (The details are highly nontrivial, especially if control flow can be affected, but they can be solved; see, e.g., Correia et al. [19] for a practical technique.) With this change, we can use a crash-tolerant algorithm, as long as we can find a setting of k such that the failure probability we are willing to tolerate is higher

than the probability of memory corruption causing *either* all k computations to return identical incorrect results *or* more crashes than the crash-tolerant algorithm can handle. Notice that increasing k will reduce the probability of the former but increase the probability of the latter, so there is not always a suitable setting for k.

Summary: If a small probability of failure is acceptable, memory corruption can be reduced to crash faults.

I. What if the system is asynchronous?

The next possible reason for rejecting our trivial solutions is that the system is asynchronous. This is the first of two issues that really do substantially increase the complexity of the problem. One important consequence of asynchrony is that nodes can no longer tell reliably whether another node is faulty or is just very slow. Because of this, solutions, such as Paxos [38], typically require a majority of the nodes to be correct: intuitively, a group of $N - f$ nodes must be able to decide without hearing from the other f nodes at all, since up to f nodes may crash, but two non-overlapping groups must not be allowed to both decide, so we need $2 \cdot (N - f) > N$, or $N > 2f$. This lower bound disappears if nodes are given a failure detector [14, §6.1], so the inability to detect faults really is at the heart of the problem.

However, true asynchrony is rare [60] – especially in hard real-time systems, which in some sense are the antithesis of asynchronous systems. Once WCETs are available and communication is carefully scheduled, it is possible to get upper bounds on virtually any delay. Thus, in the systems we are considering here, our trivial solutions should almost never be rejected for this particular reason alone.

Summary: Asynchrony requires more complex solutions, but true asynchrony is incompatible with hard real-time systems.

J. What if nodes can equivocate?

A final reason for rejecting our trivial solutions is that nodes in the system can equivocate – in other words, that a faulty node can give different and conflicting information to two correct nodes. This is an obvious threat to consistency, and handling it increases complexity quite a bit. In general, consensus in this model requires $N > 3f$ nodes if asynchrony is present (Example: PBFT [11]), and $N > 2f$ nodes otherwise (Example: Abraham et al. [2]). The point that equivocation is at the heart of the problem has been made in several prior works, e.g., [9], [18], [46], [54]; another way to see it is to consider TrInc [40], which shows that a small trusted-hardware gadget that prevents otherwise Byzantine nodes from equivocating is enough to drop the bound for asynchronous systems back to $N > 2f$.

Equivocation is a common complication of the Byzantine model: once a node is compromised by an adversary, it can send conflicting messages to other nodes. However, even with Byzantine faults, equivocation in a synchronous system with a reliable network is a serious problem only if a) communication is through unicast channels, and b) messages are not signed. If communication is via a broadcast channel, a faulty node
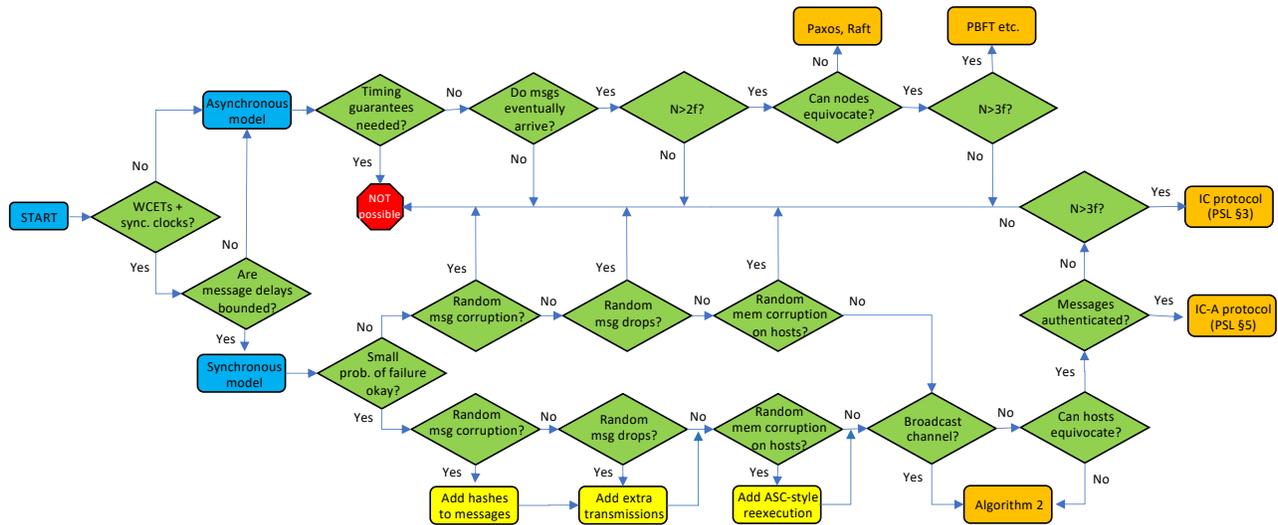
Fig. 1. A relevant part of the design space for fault-tolerant consensus. (The figure is not meant to be exhaustive.) The orange boxes show the four widely known "difficult" classes of algorithms: synchronous with authentication (IC [49, §5]) and without (IC-A [49, §3]), and asynchronous with Byzantine faults (PBFT [11]) and crash faults (Paxos [38] or Raft [47]). The fifth class, which T-Cons trivially solves, is less widely known, but we argue that real-time/embedded systems, in particular, are often able to use it.

Technique	Sect.	Reduction
Redundant messages	II.F	Prob. msg drops → "Reliable network"
Hashes	II.G	Prob. msg corruption → Prob. drops
Repeated comp. [19]	II.H	Prob. memory corrupt. → Crash faults

TABLE I

cannot equivocate because all the other nodes will have the same view of the messages it sends. If messages are signed, we can use a construction analogous to the SM protocol [39, §4]: correct nodes can add their signature to messages they receive and forward them to the other nodes; once a message has collected $f + 1$ signatures, it must have been seen and forwarded by at least one correct node, so the correct nodes can be sure that they each have seen it – and, if this is done for all messages, that they have a consistent view of the messages each node has sent.

It may seem odd to see a solution for consensus when a majority of the nodes can be faulty. The confusion arises because, in the literature, consensus is often used for state-machine replication: the replicas agree on a request ordering, then they execute the requests and send the results back to the client. If a client receives different responses from different replicas, it generally cannot tell which response is correct and must go with the majority; thus, with state-machine replication, we do need at least $2f + 1$ replicas to ensure that the majority result is correct, independent of any lower bounds from the consensus stage.

How common are real-time systems with unicast channels and no signatures? Many embedded systems do have a link-layer broadcast primitive – either because they use a bus topology directly (CAN bus, etc.) or because they use a network that supports broadcast, such as switched Ethernet. And cryptographic signatures are not an unreasonable assumption, even for embedded systems – there are fast implementations on embedded CPUs [42], [57] and many different hardware accelerators [50], [55], [61].

Summary: Equivocation does increase complexity, but only in combination with unicast channels and without authentication.

K. Summary

Figure 1 summarizes the points we have made in this section, and Table I summarizes the reductions we used. Although consensus has a reputation for being a complex and challenging problem, this is true only if: 1) the system is asynchronous, 2) nodes can equivocate and there is neither a broadcast channel nor message authentication, or 3) there is message loss, message corruption, or memory corruption, and we insist on perfect guarantees and cannot accept even a small probability of failure. None of these scenarios seem particularly likely for real-time or embedded systems.

III. CASE STUDY: THE GBB PROTOCOL

Our first case study is a protocol by Gujarati et al. [27] that appeared at RTAS 2020. For brevity, we will refer to this protocol as GBB below.

A. Problem statement

We begin by reviewing the problem as defined in [27]. There are N nodes that are connected via Ethernet switches. Each node i starts with a local input v_i and must eventually decide on a value. The goal is to provide one of the following two properties:

- **Strong correctness:** If $v_i = v$ for a majority of the nodes, then a majority of the nodes will decide on v; otherwise the nodes may decide on any value, or on $\perp$.
- **Weak correctness:** If v is the most common input value (with ties broken deterministically), more nodes will decide on v than on any other value, including $\perp$.

This problem is a relaxed version of consensus, in two ways: (1) it does not demand that *all* the *correct* nodes decide on

Algorithm 3 The GBB algorithm, from [27]

```
 1: procedure INITIALIZATION
 2:     EIG_i.addRoot(⟨ε, v_i⟩)
 3: procedure ROUND(r)
 4:     for all ⟨α, v⟩ ∈ EIG_i.nodes s.t. |α| = r − 1 do
 5:         if Π_i ∉ α ∧ v ≠ ⊥ then
 6:             send ⟨α, v⟩ to all processes in Π \ {Π_i}
 7:     for all ⟨α, v⟩ ∈ EIG_i.nodes do
 8:         if |α| = r − 1 then
 9:             for all Π_j ∈ Π s.t. Π_j ∉ α do
10:                 if Π_i = Π_j then
11:                     EIG_i.addChild(⟨α, v⟩, ⟨αΠ_j, v⟩)
12:                 else if ⟨α, v'⟩ is received from Π_j then
13:                     EIG_i.addChild(⟨α, v⟩, ⟨αΠ_j, v'⟩)
14:                 else
15:                     EIG_i.addChild(⟨α, v⟩, ⟨αΠ_j, ⊥⟩)
16:     if r ≠ N_r then return
17:     for all ⟨α, v⟩ ∈ EIG_i.nodes from |α| = N_r − 1 to |α| = 1 do
18:         candidates = ∅, v_{majority} = ⊥
19:         for all ⟨αΠ_j, v'⟩ ∈ EIG_i.getChildren(⟨α, v⟩) do
20:             if v' ≠ ⊥ then
21:                 candidates = candidates ∪ {v'}
22:         if candidates ≠ ∅ then
23:             v_{majority} = simpleMajority(candidates)
24:         EIG_i.updateValue(⟨α, v⟩, v_{majority})
25:         if α = Π_k then
26:             V_i[k] ← v_{majority}
```

the same value, but rather a majority of the nodes (somewhat analogous to almost-everywhere agreement [24]), regardless of whether they are correct, and (2) strong correctness does not make any demands at all when no value is proposed by a majority.

The paper assumes that nodes can crash or suffer random memory corruption; the system is synchronous, and messages can be corrupted in the network. There are no assumptions about authentication, so we assume that signatures are not available. Unicast and broadcast are not mentioned specifically, but the paper does assume an Ethernet network, which supports link-layer broadcast.

B. Original solution

The GBB paper proposes a solution (Algorithm 3) that is a modification of the Exponential Information Gathering (EIG) trees from [8] (which in turn follows [49, §3]), a consensus protocol for synchronous systems with Byzantine faults. Briefly, the EIG protocol consists of two steps: 1) The nodes compute a vector V, whose elements are the values v_i that each node has proposed, or $⊥$ if nodes fail or misbehave in a certain way; the protocol guarantees that all correct nodes compute the same vector, as long as $N > 3f$. Then 2) each node uses a deterministic function f to map V to a single element $f(V) ≠ ⊥$, and then decides that value.

The computation in the first step proceeds in several rounds. In the first round, each node i sends a tuple $⟨ε, v_i⟩$ ("my value is v_i") to every other node. In each of the following rounds, each node takes the tuples it has received in the previous round, appends the sender's ID to the first element of each, and sends the resulting tuples to each other node. Thus, if j received $⟨ε, v_i⟩$ from i in the first round, it distributes tuples $⟨i, v_i⟩$

("i told me that its value was v_i") in the second round; if k received this tuple, it then distributes $⟨ij, v_i⟩$ ("j told me that i told j that i's value was v_i") in the next round, etc.

Each node i organizes the tuples it receives in a tree, in which $⟨ε, v_i⟩$ is the root and, for any string $α$, $⟨αj, ...⟩$ is a child of $⟨α, ...⟩$. If i has received no tuple $⟨α, v⟩$ in round $|α|$, it fills in $⟨α, ⊥⟩$. After the last round, each node walks its tree from the leaves to the root; for each interior node $⟨α, v⟩$ it encounters, it replaces v with a function of the values in the direct children. The choice of this function is one of two major differences between EIG and GBB: EIG uses uses any value that has been received at least $N − |α| − f$, whereas GBB simply uses the majority of the non-$⊥$ values. If no suitable value is found, both protocols use $⊥$.

At the end of this process, the direct children of the root on node n_i contain a value $v_{i,j}$ for each node n_j. EIG guarantees that 1) all the correct nodes compute the same vector, and 2) if n_k is correct, then $v_{i,k} = v_k$. Finally, the nodes apply a deterministic function to the v_i to pick a single value to decide; EIG does not prescribe a particular function, but GBB uses the majority of the non-$⊥$ values.

C. Does GBB tolerate Byzantine faults?

Although the GBB paper focuses mainly on benign corruptions, it describes the goal as "tolerating [...] Byzantine errors" [27, §I]. The choice of the more general Byzantine fault model, as opposed to simple probabilistic faults, has a number of problematic consequences. The first of these is that – strictly speaking – the problem cannot be solved at all: *no algorithm* (including GBB) can guarantee either strong or weak correctness if even a single node might be Byzantine. We prove this claim with the following two theorems.

Theorem 1. *No algorithm can guarantee GBB's notion of strong correctness in the presence of at least one Byzantine node.*

Proof. Suppose some algorithm A does provide strong correctness when one node – say, B – is Byzantine. (If there are other Byzantine nodes, they can just execute the algorithm correctly; recall that Byzantine nodes can have any behavior, including, as a special case, the correct behavior.) Consider two executions e_1 and e_2; in both executions, $⌊N/2⌋$ of the other nodes have input X, and $⌈N/2 − 1⌉$ of the other nodes have input Y; B has input X in e_1, and Y in e_2. Then strong correctness requires that a majority of the nodes decide X in e_1, and Y in e_2. But now suppose that, in e_2, B pretends that its input is X – that is, it performs the exact steps that A requires, but replaces its input with X. However, note that each node can only make decisions based on information it has locally available. Nodes cannot know "what really happened" on another node; they can only observe messages that were sent by that other node. Then, from the perspective of the other nodes, B's steps in e_1 and e_2 will be indistinguishable, and since their own circumstances are identical in both executions, these other nodes must make the same decision in both. Since B makes the same decision in both executions as well, a

Algorithm 4 The T-GBB algorithm
```
1: procedure ROUND(r)
2:     decide(v_i)
```

Algorithm 5 The S-GBB algorithm
```
1: procedure INITIALIZATION
2:     V[k] = (k == i) ? v_i : ⊥
3: procedure ROUND(r)
4:     broadcast(⟨V, H(V)⟩)          ▷ H(·) is the hash function from §II-G
5:     if ⟨V′, h⟩ is received from n_k and H(V′) = h then
6:         for each k : V[k] = ⊥ and V′[k] ≠ ⊥ do
7:             V[k] ← V′[k]
8:     if r = N_r then decide(simpleMajority({V[i] | V[i] ≠ ⊥}))
```

majority of the nodes must make the *same* decision in both executions, and since the correct decision (which GBB defines based on the majority of the *inputs*) was X in e_1 and Y in e_2, one of these decisions must be wrong. □

Theorem 2. *No algorithm can guarantee GBB's notion of weak correctness in the presence of at least one Byzantine node.*

Proof. The proof is largely analogous to the proof of Theorem 1, except that in e_1, weak correctness demands that more nodes decide on X than on any other value, including $\perp$; in e_2, it demands that more nodes decide on Y than on any other value, including $\perp$. As before, both cannot be true in the same execution, but B can make the two executions indistinguishable by simulating A in e_2 with its input value replaced with X. Thus, no algorithm can make the correct decision in both executions. □

These results may be surprising: the problem does not seem all that different from the consensus problem in Section II-B, which does have solutions in the Byzantine model. The difference is that consensus defines correctness in terms of what nodes *do* (proposing a value), which other nodes can observe and respond to, and not in terms of private inputs that are known to only one particular node [45]. Since a Byzantine node can always tell lies about its inputs, a goal that is based on inputs alone would be difficult to achieve.

D. A trivial solution

Of course, the fact that GBB cannot tolerate Byzantine faults does not mean that it is not useful; from the context, it is clear that the real goal is to tolerate specific errors (crashes, memory corruption, and message corruption) with high probability. We next discuss whether the proposed solution, a variant of EIG trees, is a good fit for this goal.

However, as defined in [27, §II.B], the problem has a trivial solution (Algorithm 4) – every node can simply decide its local value! This approach, which we refer to as T-GBB, is already enough to provide GBB's notion of strong correctness: if there is an input v that a majority of the nodes have received, then these nodes will decide v and form the required majority, and if there is no majority input, strong correctness makes no demands. Since this algorithm sends no messages and performs no computation, none of the assumed probabilistic faults can occur, so it is perfectly reliable.

We assume that this was not the goal, so, in the following, we assume that a slightly stronger form of agreement is required: at least in a fault-free execution, all the correct nodes should decide on the *same* value. The GBB algorithm does have this property.

E. A simple alternative

With this addition, a Byzantine-tolerant protocol like EIG trees seems like a natural choice, since the fault model includes non-crash faults and Byzantine-tolerant protocols can handle these. However, probabilistic faults are only a small subset of the entire set of Byzantine faults, and an "easy" subset at that – as we have argued in Section II-J, the truly difficult faults are the ones where nodes equivocate, which can occur only by accident in the probabilistic model.

None of the other complicating factors are present, either: the network can drop or corrupt messages, but the goal is to give probabilistic guarantees, so we can use hash functions to convert corruptions to drops (Section II-G) and retransmissions to mask drops (Section II-F). Nodes can suffer memory corruption, but we can convert most of these to crashes by executing computations more than once (Section II-H). Finally, the system is synchronous, so the biggest potential source of complexity is absent entirely.

Thus, it seems reasonable to consider a simpler replacement. Our proposal is Algorithm 5, which we will refer to as S-GBB. This is a variant of our trivial solution for consensus (Algorithm 2), but with multiple rounds to account for message drops, as well as with hash functions added (see §II-G), and with f instantiated with the simple majority, as required by GBB's notion of strong correctness. To handle memory corruption, the computations would need to be reexecuted a few times, as discussed in §II-H; we omit this here for clarity. Since [27] considers an Ethernet network, the algorithm takes advantage of Ethernet's link-layer broadcast primitive.

F. Overhead

We now compare GBB and S-GBB in terms of cost. This is an important factor in choosing fault-tolerance algorithms; BFT solutions, in particular, are notoriously expensive. At first glance, S-GBB is more efficient than GBB simply because each node sends only one (broadcast) message per round, whereas GBB sends unicast messages; thus, for R rounds, the per-node message complexity appears to be $O(R)$ for S-GBB and $O(N \cdot R)$ for GBB.

However, the actual cost difference is much higher than that. Vanilla message complexity obscures the fact that EIG sends a lot more information: each message contains an entire level of the EIG tree. Recall that, on a node n_i, the first level contains n_i's own input, the second level says, for each node n_j, what n_j told n_i its input was, the third level says, for each pair of nodes (n_j, n_k), what n_k told n_j told n_i its input was, etc.
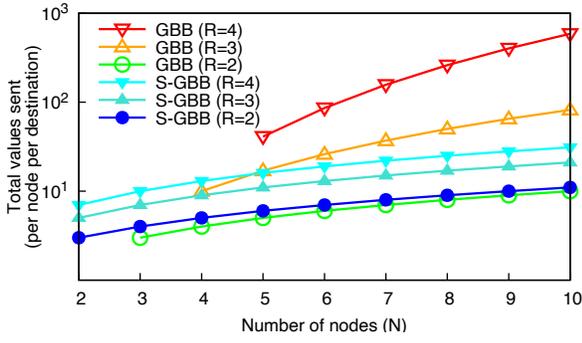
Fig. 2. Total number of values each node sends to each other node over the network, when R rounds of GBB/S-GBB are executed.

Fig. 3. Values stored in memory by GBB and S-GBB, when R rounds are executed. S-GBB's memory requirements do not depend on R.

In general, level r contains a vertex for each sequence of r distinct nodeIDs that ends in n_i. There are $\frac{(N-1)!}{(N-r)!}$ such labels, and for each label, EIG (and, thus, GBB) sends at least one value to each other node. Thus, if GBB runs for R rounds, the total number of values each node sends is actually

$$\sum_{r=1}^{R}(N-1)\cdot\frac{(N-1)!}{(N-r)!}$$

which is $O(N^R)$, whereas S-GBB sends one value in the first round (if we omit the $\perp$ values), and N values in each subsequent round, so each node sends $1+N\cdot(R-1)$ values, which is $O(N\cdot R)$. Figure 2 shows, for different values of N and R, how many values each node sends to each other node. (To be fair, we ignore the fact that S-GBB uses link-layer broadcasts, which, in principle, GBB could do as well.) The figure shows that the S-GBB's lower complexity can make a substantial difference, especially for larger numbers of rounds.

Figure 3 compares the two algorithms in terms of their storage requirements. GBB must remember each of the $\sum_{r=1}^{R}(N-1)\cdot\frac{(N-1)!}{(N-r)!}$ values it receives, so its storage complexity is $O(N^R)$. In contrast, S-GBB stores only a single estimate for the value of each node, so it simply requires N values, regardless of R. In an embedded system, this can be a substantial benefit: for instance, with $N=8$ and $R=4$, GBB stores 1,100 values, while S-GBB stores only eight.

G. How to assess reliability?

Another obvious factor in choosing a fault-tolerance algorithm is reliability – the probability that the algorithm will succeed.

Since GBB assumes probabilistic faults, we can, in principle, compute the probability of success using probabilistic model checking. In essence, this approach considers all possible executions of the algorithm under all possible failure scenarios, computes the probability of each execution, and adds up the probabilities for the executions where the algorithm succeeds. The GBB paper rejects this approach, essentially because the space of possible faults is enormous: nodes could crash at every step, every single message could potentially be lost, etc. For instance, suppose we run S-GBB with $N=4$ nodes and $R=3$ rounds, and suppose we let each node
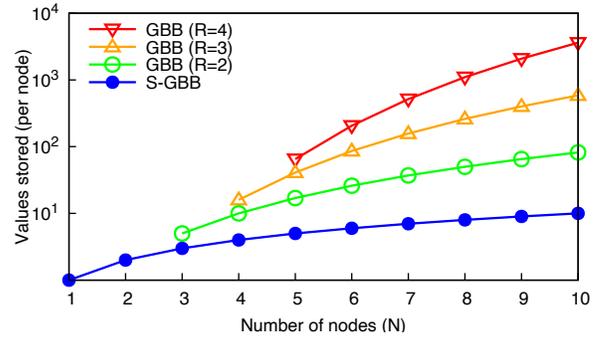
pick one of four possible inputs (since there could be at most four different ones). Then there are $4^4=256$ possible runs; in each run, each node can crash in each round, or not at all, so there are $4^4=256$ crash scenarios; and there are up to $N\cdot(N-1)\cdot R=36$ opportunities for message drops. Thus, if we ignore for the moment that some messages cannot be lost because their sender has already crashed, there are $4^4\cdot4^4\cdot2^{36}\approx4.5\cdot10^{15}$ scenarios for the model checker to consider – a staggering number!

However, while it is true that model checking *in general* suffers from state explosion, this particular setting happens to be relatively easy, as long as we can accept an upper bound. The reason is that faults are relatively rare. For instance, suppose crashes and message losses occur independently, and we use a crash probability of $p_{\text{crash}}=10^{-8}$ and a loss probability of $p_{\text{drop}}=10^{-3}$. Then the scenario where one node crashes in the last round *and* four of the 33 messages are lost has probability $\frac{1}{4}^4\cdot(1-p_{\text{crash}})^{11}\cdot p_{\text{crash}}\cdot(1-p_{\text{drop}})^{29}\cdot p_{\text{drop}}^4\approx3.79\cdot10^{-23}$. If we are only interested in certifying a failure probability of less than, say, 10^{-15}, it is not clear that we need to consider such scenarios (or ones that are even less likely) at all – most of the probability mass will be concentrated in a small number of relatively likely scenarios.

We can get a precise range for the failure probability as follows. We maintain a set of execution prefixes and associated probabilities, which is initially set to the empty prefix and probability one. Then we repeatedly pick the most likely prefix that has not yet terminated and execute one step of the algorithm; if the step is probabilistic, we "split" the current prefix into multiple prefixes. Thus, at each moment, we have lower and upper bounds on the probabilities of success and failure: for the lower bounds, we can add up the probabilities of the prefixes that have terminated with success and failure, respectively, and for the upper bounds, we can add to each the probabilities of the "undecided" prefixes that are still active. We can terminate this process at any moment, once the upper and lower bounds are close enough for our purposes. Figure 4 shows how the cumulative probability of the undecided prefixes evolves as more and more executions are evaluated; after as few as 973,824 executions, the undecided prefixes account for less than a probability of 10^{-15}. Examining this
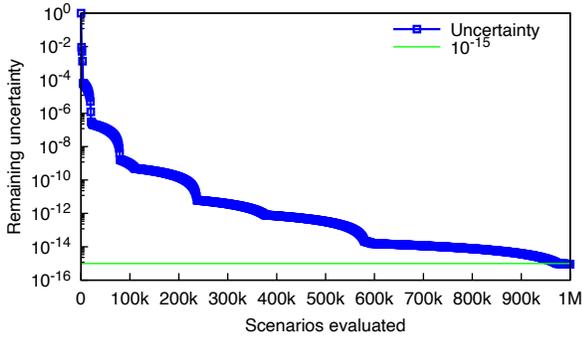
Fig. 4. Remaining uncertainty after a certain number of scenarios have been evaluated with probabilistic model-checking.

Fig. 5. Failure probabilities, as a function of the number of nodes and rounds. The error bars show lower and upper bounds.

many executions is perfectly feasible on modern hardware, and many model checkers would be able to do even better, e.g., by considering symmetry.

The GBB paper instead presents a closed-form analysis that involves about 5.5 pages of complex math, as well as a custom, 53-line algorithm. This approach is labor-intensive, as the manual analysis would have to be repeated for every new algorithm or variant, and it almost inevitably involves upper-bounding the more complicated probabilities, which is a source of potential mistakes. For instance, in the analysis of case #5, the GBB paper observes that it is "impossible to estimate" the probability of failure after message corruption "without knowing the exact contents of the corrupted messages"; it therefore chooses a worst-case analysis, which assumed that all corruptions turn the message into a special Incorrect value [27, §IV.D.5]. But this is not actually the worst case! Consider a situation with a narrow majority, analogous to Section III-C, perhaps one where five nodes have input A and four nodes have input B. If a corruption flips an A into a B, this could cause nodes to decide on B and thus violate weak correctness, whereas, if A can only flip to Incorrect, weak correctness could still hold, since the Incorrect value will be in a small minority and will thus almost certainly be ignored by the algorithm. As a result, the computed probability is not necessarily an upper bound.

This kind of issue is not uncommon when analyzing non-crash behavior manually – there are almost always lots of cases, and it is rarely obvious what the worst case is. In light of this difficulty, a mechanized approach, such as probabilistic model checking, seems like the safer choice.

H. Reliability

To get a side-by-side comparison of GBB and S-GBB in terms of reliability, we evaluated both with probabilistic model checking, using the approach we sketched above. Where possible, we use assumptions that are comparable to [27]: we assume that nodes crash with probability 10^{-8} in each round (and do not recover before the protocol terminates); we assume that messages experience corruption with probability 10^{-3} and are dropped by the Ethernet link layer unless the corruption results in a CRC32 collision, which maps to an effective drop rate of 10^{-3} (for practical purposes) and, for GBB, a
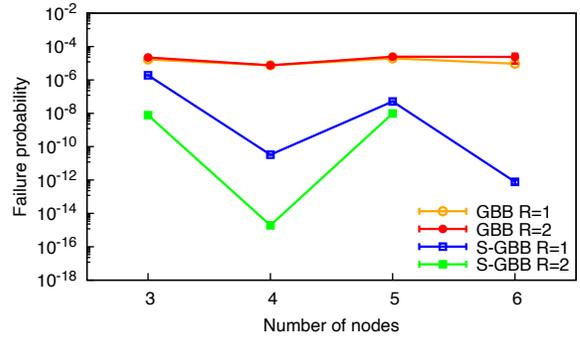
corruption rate of $2.33 \cdot 10^{-13}$. For S-GBB, we assume a SHA-1 hash, whose collision probability of $6.8 \cdot 10^{-49}$ seems small enough to ignore; similarly, GBB assumes a host corruption rate of 10^{-5}, but if we execute S-GBB's very simple update step on ten copies of the vector and crash the node unless seven of the copies agree, we end up with an extra crash rate of $\approx 10^{-20}$ and a host corruption rate of less than 10^{-35}; both again seem low enough to ignore. We assume that corruption does not occur before S-GBB has had a chance to make copies of the node's local input value.

Our numbers are different from the ones reported in [27, Fig.6], for at least two reasons. First, in conversations with the authors of [27], we discovered that the experiment in [27, §V.B] was done with a different notion of strong consistency than the one the paper had defined in §II.B: in the experiment, each node received a different input value, and a run was considered a success when a majority of the nodes was able to correctly infer a majority of the private input values, without considering what each node actually decided. Here, we use the paper's original definition from §II.B instead; to ensure that majorities can form, we assume that each node randomly receives one of two possible inputs, with equal probability. Second, rather than assuming that memory corruptions result in a special Incorrect value, we randomly flip one of the values in the EIG tree.

Figure 5 shows the failure probabilities of GBB and S-GBB as a function of the number of nodes and rounds. GBB's failure rate is higher than S-GBB's; its most common source of failure is host corruption, which can, in the common situation where both inputs are about equally prevalent, flip the majority the wrong way. Increasing the number of nodes N does not help much because a) the number of nodes with a given input follows a Binomial distribution, which has most of its probability mass concentrated around the problematic $\frac{N}{2}$ point, and b) increasing N also increases the expected number of corruptions. For S-GBB, the most common source of failures is message drops preventing some of the nodes from learning the inputs of certain other nodes. This becomes less likely if we increase the number of nodes N and/or the number of rounds: a node n_i learns the input of another node n_j as long as there is *at least one* path from n_i to n_j with no drops or corruptions. The curious dips at even numbers of nodes

Algorithm		Synchr.	Topology	Drops	Assumptions Mem.corr.	Crashes	Adversary	Auth.	Goals		Costs (per node) Communic.	Storage
T-Cons	Alg.2	Sync	Broadcast	No	No	Yes	No	No	Consensus	Perfect	$O(N)$	$O(N)$
GBB [27]	Alg.3	Sync	Broadcast*	Prob.	Prob.	Prob.	No	No*	GBB-SC	Prob.	$O(N^R)$	$O(N^R)$
T-GBB	Alg.4	Sync	(any)	(any)	(any)	(any)	(any)	(any)	GBB-SC	Perfect	Zero	Zero
S-GBB	Alg.5	Sync	Broadcast	Prob.	Prob.	Prob.	No	No	Consensus	Prob.	$O(N^2 \cdot R)$	$O(N)$

<div align="center">TABLE II</div>

happen because of the way [27] defines strong correctness: the nodes are free to decide on anything at all unless there is an input value that occurs at least $\lfloor N/2 \rfloor + 1$ times. For even values of N, the most common case is that the two inputs occur equally often, and in this situation strong correctness is trivially satisfied, no matter what the nodes decide.

I. Summary

The main reason GBB "overpays" for fault tolerance is a disconnect between the faults it assumes and the fault model it uses: all random corruptions are Byzantine faults, but the reverse is not true. In particular, the Byzantine model assumes that faulty nodes might take the worst possible action *every single time*, instead of just with a small probability, and this is a major source of complexity in Byzantine-tolerant algorithms. In fact, as we have shown in Section III-C, it is *impossible* to solve the problem the paper motivates when even a single Byzantine node is present! Solutions do exist for the probabilistic faults that [27] focuses on, but, as T-GBB and S-GBB demonstrate, they can be much simpler than GBB.

EIG trees are both too strong and too weak for this setting. They are too strong because they can tolerate the entire set of Byzantine faults, when only a small subset is needed here, and they solve consensus, when the goal is only to achieve a form of almost-everywhere agreement. And they are too weak because they assume that the number of faulty nodes cannot reach $N/3$, which is not true in this setting. When this assumption is violated, EIG trees can actually make matters worse, since nodes will follow (potentially faulty) quorums of a certain size, even if the correct value reached them as well.

Table II shows a comparison of the four candidate solutions (T-Cons, GBB, T-GBB, and S-GBB) we have discussed here. (Properties with asterisks are inferred; GBB-SC means strong correctness as defined in [27].) S-GBB offers substantial benefits over GBB: it sends less data and consumes less storage, it is more reliable, it should be easier to implement, and it is easier to analyze.

IV. CASE STUDY: RT-BYZCAST

Our second case study is a protocol by Kozhaya et al. [37] that provides a form of Byzantine-tolerant broadcast.

A. Problem statement

There are N nodes that are connected by a network. The goal is to implement a broadcast primitive called RTBRB that has the following five properties:

- **RTBRB-Validity:** If a correct process broadcasts m, then some correct process eventually delivers m.
- **RTBRB-No duplication:** Every correct process delivers each message at most once.

- **RTBRB-Integrity:** If some correct process delivers a message m with sender n_i, and node n_i is correct, then m was previously broadcast by n_i.
- **RTBRB-Agreement:** If some message m is delivered by any correct process, then every correct process eventually delivers m.
- **RTBRB-Timeliness:** There exists a known Δ such that, if a correct process broadcasts m at real-time t, no correct process delivers m after real time $t + \Delta$.

This problem is a variant of what we have called broadcast in Section II-A; the properties are stated slightly differently, and there is an additional timeliness requirement.

The paper assumes that up to $f = \lfloor \frac{n-1}{3} \rfloor$ nodes can be compromised by an adversary. The system is synchronous; messages are authenticated and can be randomly dropped in the network. Since the latter precludes a perfect implementation of RTBRB-Timeliness, an additional requirement is that nodes crash themselves if they appear to be disconnected from too many other nodes.

B. Original solution

Algorithm 6 shows the original RT-ByzCast algorithm from [37]; for brevity, we have omitted the code for the *proof-of-life*, *aggregate-sig*, and *deliver-message* functions.

Briefly, the algorithm works as follows. When a node wishes to broadcast a message m, it signs m and sends a `RTBRB-broadcast` message to every other node. When a node sees this message, it responds with an `Echo` message that includes a signature of its own. The echo signatures are aggregated in the R_{echo} set on each node; once a node has accumulated a quorum of at least $2f + 1$ signatures, it sends a `Deliver` message that includes these signatures to each other node. `Echo` and `Deliver` messages are retransmitted for some time, in case they are lost. If a node has no messages to send, it sends a heartbeat message; if a node fails to accumulate signatures from a quorum on either broadcasts or heartbeats, it crashes itself.

C. A simple alternative

Since the paper assumes that nodes can potentially be compromised by an adversary, its fault model encompasses the entire set of Byzantine faults, so, at first glance, the complexity of the original solution seems justified and the upper bound on the number of faulty nodes seems necessary. Notice that, since messages can be dropped and we insist on perfect guarantees, the limit for asynchronous systems applies, even though the system itself is synchronous.

However, in the system model in §II.A, the paper mentions almost in passing that nodes are assumed to be connected by links, and that links "can abstract a physical bus or a dedicated

Algorithm 6 The RT-ByzCast algorithm, from [37]

1: **Init:** Msg[] ^{n} [] $^n = \emptyset$
2: **Execute** *proof-of-life*(R);
3: **upon event** $< p_i$ wants to broadcast a value $v >$ **do**
4:     **Execute** *proof-of-life* function in piggyback mode
5:     **Initialize** $R_{echo}(p_i, r, v) = \emptyset$
6:     **Send** periodically starting from the current round
        RTBRB-broadcast $((p_i, r_{current}, v); \Phi_{p_i})$ to all $p \in \Pi$
7:
8: **upon event** $<$receive RTBRB-broadcast() $(p_i, r', v); \Phi_{p_i})$ in
    round $r - 1 \geq r'$ for the first time$>$ **do**
9:     **Execute** *proof-of-life* function in piggyback mode
10:     **Initialize** $R^{p_j}_{echo}(p_i, r, v) = p_i; sigs = \Phi_{p_j}$
11:     **Send** Echo$_{p_j}((p_i, r', v; \Phi_{p_i}); sigs)$ to all $p \in \Pi$ at rounds $\geq r$
12:
@process p_j:
13: **upon event** $<$receive Echo$_{p_k}((p_i, r', v; \Phi_{p_i}); \Phi_{p_x}, \ldots, \Phi_{p_z})$ at
    round r **do**
14:     **if** p_j is not sending any Echo() **then**
15:         **Set** $sigs = aggregate\text{-}sig_{p_j}(v, p_i, \Phi_{p_x} \ldots \Phi_{p_z}, p_k)$
16:         **if** p_j has not already delivered a message relative to p_i **then**
17:             **Execute** *proof-of-life* function in piggyback mode
18:             **Initialize** $R_{echo}(p_i, r + 1, v) = \emptyset$
19:             **If** $sigs \leq 2f$ **then**
20:                 **Send** at the beginning of every cycle (as of round
                $r + 1$ onward) Echo$_{p_j}((p_i, r + 1, v; \Phi_{p_i}); \Phi_{p_j})$ if $k \neq j$
21:             **end if**
22:         **end if**
23:         **If** $sigs > 2f$ (for the first time) **then**
24:             **Set** $R_{echo}(p_i, r, v) = sigs$
25:             **Execute** *deliver-message*$(p_i, v, sigs)$
26:         **end if**
27:     **end if**
28:     **if** p_j is sending an Echo$_{p_j}((p_i, r', v; \Phi_{p_i}); \ldots)$ **then**
29:         **Set** $sigs = aggregate\text{-}sig_{p_j}(v, p_i, \Phi_{p_x} \ldots \Phi_{p_z}, p_k)$
30:         **If** $sigs > 2f$ (for the first time) **then**
31:             **Set** $R_{echo}(p_i, r, v) = sigs$
32:             **Execute** *deliver-message*$(p_i, v, sigs)$
33:         **end if**
34:         **If** $sigs \leq 2f \wedge k = j$ **then**
35:             **Set** $R_{echo}(p_i, r, v) = sigs$
36:         **end if**
37:         **If** $sigs \leq 2f$ **then**
38:             **Send** at the beginning of every cycle (as of round $r + 1$
              onward) Echo$_{p_k}((p_i, r', v; \Phi_{p_i}); sigs)$ to all $p \in \Pi$
39:         **end if**
40:     **end if**
41:     **if** p_j is sending Echo$_{p_j}((p_i, r'', v'; \Phi_{p_i}); *) : v' \neq v$ **then**
42:         **Set** $sigs = aggregate\text{-}sig_{p_j}(v', p_i, \Phi_{p_x} \ldots \Phi_{p_z}, p_k)$
43:         **If** $sigs > 2f$ (for the first time) **then**
44:             **Set** $R_{echo}(p_i, r, v) = sigs$
45:             **Execute** *deliver-message*$(p_i, v', sigs)$
46:         **end if**
47:     **end if**
48:
49: **upon event** $<$receive Deliver$_{p_k}((p_i, v, sigs); \Phi_{p_x} \ldots \Phi_{p_z})$ at
    round r **do**
50:     **if** $((p_i, v)$ is not delivered yet **then**
51:         **Deliver** v
52:         **Stop** sending any Echo()
53:         **Initialize** set $R_{deliver}(p_i, r) = \{p_x, \ldots, p_z\}$
54:     **else**
55:         $R_{deliver}(p_i, r') = R_{deliver}(p_i, r') \bigcup \{p_x, \ldots, p_z\}$.
56:     **end if**
57:     **Send** Deliver$_{p_j}((p_i, v, sigs); signatures)$ to all $p \in \Pi$ at
    every round in $[r + 1, r + 1 + 2R]$, $signatures$ contains the
    signatures of all processes in $R_{deliver}(p_i, \ldots)$.
58:     **Execute** same commands as lines 5-8 of *deliver-message*$_{p_j}(\ldots)$
59:

Algorithm 7 The S-ByzCast algorithm

1: **procedure** INITIALIZATION
2:     seen $= \emptyset$
3: **procedure** RTBRB-BROADCAST(m)
4:     **for** $i = 1 \ldots k$ **do**
5:         broadcast$(i, m, now, \sigma_i(i, m, now))$
6: **procedure** ROUND(r)
7:     **for each** received $msg := (j, m, r, s)$ **do**
8:         **if** signature s is valid **then**
9:             **if** $\neg \exists m', s' : (j, m', r, s') \in seen$ **then**
10:                 seen = seen $\cup \{msg\}$
11:                 RTBRB-deliver(m)

network link/path". If the network does indeed consist of a physical bus, this provides a broadcast channel and thus removes the adversary's ability to equivocate, which, as we have discussed in Section II-J, is a major source of complexity.

With a *reliable* broadcast channel, the solution would be trivial and essentially equivalent to Algorithm 1. However, the assumption here is that the network can randomly drop messages, and since the RTBRB guarantees are deterministic, it seems that one still cannot avoid collecting quora. However, we note that RT-ByzCast's effective guarantees are probabilistic as well, since a) a sequence of random message drops can cause correct nodes to crash themselves, and b) too many crashes can cause the number of live nodes to fall below $3f+1$, at which point the Byzantine nodes can, by falling silent, cause *all* of the remaining correct nodes to crash themselves as well. If this cataclysmic scenario is acceptable, provided that its probability is small enough, a small chance (say, 10^{-15}) of a loss of RTBRB-Agreement may be acceptable as well.

For this specific case (bus topology, probabilistic guarantees), we can suggest a much simpler alternative (Algorithm 7), which we call S-ByzCast. This algorithm simply 1) signs messages and broadcasts them k times, and 2) discards duplicates, messages that are not properly signed, and messages that conflict with another message sent by the same node in the same round. Validity holds with high probability if k is large enough; integrity is ensured by the signatures; agreement holds because of the broadcast channel (which delivers the message either to all the correct nodes, or to none of them); and timeliness holds because the system is synchronous. This algorithm is much simpler, and uses fewer messages; moreover, it no longer has a nontrivial bound on f and thus works with *any* number of Byzantine nodes.

D. Summary

If the system has a bus topology and can accept probabilistic guarantees, RT-ByzCast "overpays" for fault tolerance by not taking advantage of the natural broadcast primitive a bus provides, and by treating the bus as a collection of point-to-point links instead. As a result, the algorithm is more complex than it would need to be in that setting, it uses more messages, and it can tolerate only a limited number of Byzantine faults.

V. Related Work

Benign fault tolerance: The question of how to build safe, reliable, and fault-tolerant distributed systems has been studied in great detail by several communities, including distributed systems, real-time systems, and controller design. Existing solutions include replication protocols for asynchronous distributed systems like Paxos [38], Remus [21], and Raft [47]; fault-tolerant real-time systems, like Mars [34] and DeCo-RAM [6]; and fault-tolerant and/or reconfigurable control systems [63]. Most of this work has considered various types of "benign" faults, such as hardware defects, software bugs, or electromagnetic interference, and thus does not need the full complexity of Byzantine fault tolerance. Correia et al. [19] introduced a way to reduce corruption errors to crashes, as discussed in Section II-H.

Agreement problems: The theoretical underpinnings of broadcast and consensus are well understood. Failure detectors [13], [14] can capture the information that is needed to solve these problems; this concept was originally introduced for benign faults but was later extended to the Byzantine setting [29], [32]. There is also a rich literature on lower bounds in various settings [10], [22]–[24], [28], including the famous FLP result [25]. S-GBB is similar to the maximum information protocol by Hadzilacos [23], [28]. Pease et al. [49] proves the $N > 3f$ bound without authentication (§4) and gives protocols for both the authenticated and the unauthenticated setting (§5 and §3, respectively). The impact of equivocation on complexity was noted, e.g., in [9].

Byzantine fault tolerance: There is a rich literature on practical protocols for tolerating Byzantine faults [1], [4], [12], [16], [20], [26], [30], [36], [40], [41], [53], [58], [59], [62], and some of these protocols have been applied to distributed real-time systems (e.g., in [31], [33], [44]). And Byzantine are not uncommon: A substantial number of vulnerabilities in existing real-time distributed systems have been identified and studied, including, e.g., recent work on the on-board network in cars [15], [35], [51]. Many classical BFT protocols are unsuitable for real-time systems [52], but more recent protocols have improved in this respect [2], [5], [17], [43], and synchronous variants, such as [3], are available as well.

VI. Conclusion

In general-purpose distributed systems, fault tolerance can be a source of enormous headaches, especially when the goal is to tolerate non-crash faults. Solutions do exist, but they are often expensive, complex, and prone to subtle vulnerabilities. However, *in the particular case of real-time and embedded systems*, the problem is often much easier, or even trivial! This is because these systems are often synchronous and/or have a topology that naturally supports broadcast, and because they often have a specific reliability target and can thus accept probabilistic guarantees, as long as the probability of a failure is small enough. The theory literature has studied the benefits of synchrony as early as the 1980s, but, as recent publications show, the connection to real-time systems has either been forgotten or was never really made. We think that the community should take advantage of this connection and to adopt the "trivial" solutions whenever possible. These solutions may not be very interesting from a theoretical perspective, but they come with substantial practical advantages, such as simplicity, efficiency, and – in some cases – even higher reliability!

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. SOSP*, 2005.

[2] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren. Efficient synchronous Byzantine consensus, 2017.

[3] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *Proc. Oakland*, 2020.

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. SOSP*, 2005.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE TDSC*, 8(4):564–577, 2011.

[6] J. Balasubramanian, A. Gokhale, A. Dubey, F. Wolf, D. C. Schmidt, C. Lu, and C. Gill. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *Proc. RTAS*, 2010.

[7] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. CCS*, 1993.

[8] F. Borran and A. Schiper. A leader-free Byzantine consensus algorithm. In *Proc. ICDCN*, 2010.

[9] G. Bracha. Asynchronous Byzantine agreement protocols. *Inf. Comput.*, 75(2):130143, Nov. 1987.

[10] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824840, Oct. 1985.

[11] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI*, 1999.

[12] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Comp. Syst.*, 20(4):398–461, 2002.

[13] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[14] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225267, Mar. 1996.

[15] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proc. USENIX Security*, 2011.

[16] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. SOSP*, 2007.

[17] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. NSDI*, 2009.

[18] B. A. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, 1988.

[19] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proc. USENIX ATC*, 2012.

[20] J. A. Cowling, D. S. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. OSDI*, 2006.

[21] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. NSDI*, 2008.

[22] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[23] C. Dwork and Y. Moses. Knowledge and Common Knowledge in a Byzantine environment: Crash failures (extended abstract). In *Proc. Theoretical Aspects of Reasoning about Knowledge*, 1986.

[24] C. Dwork, D. Peleg, N. Pippenger, and E. Upfal. Fault tolerance in networks of bounded degree. In *Proc. STOC*, 1986.

[25] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374382, Apr. 1985.

[26] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proc. EuroSys*, 2010.

[27] A. Gujarati, S. Bozhko, and B. Brandenburg. Real-time replica consistency over ethernet with reliability bounds. In *Proc. RTAS*, 2020.

[28] V. Hadzilacos. A lower bound for Byzantine agreement with fail-stop processors. Technical Report TR-21-83, Harvard University.

[29] A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proc. OPODIS*, Dec. 2009.

[30] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proc. NSDI*, 2008.

[31] K. Hoyme and K. Driscoll. SAFEbus. In *Proceedings of the Digital Avionics Systems Conference (DASC)*, 1992.

[32] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[33] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[34] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, 1989.

[35] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proc. IEEE S&P*, 2010.

[36] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM TOCS*, 27(4), 2009.

[37] D. Kozhaya, J. Decouchant, and P. Esteves-Verissimo. RT-ByzCast: Byzantine-resilient real-time reliable broadcast. *IEEE Trans. Comput.*, 68(3):440454, Mar. 2019.

[38] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[39] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Prog. Lang. and Systems (TOPLAS)*, 4(3):382–401, 1982.

[40] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. NSDI*, 2009.

[41] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, 2007.

[42] Z. Liu, J. Weng, Z. Hu, and H. Seo. Efficient elliptic curve cryptography for embedded devices. *ACM Trans. Embed. Comput. Syst.*, 16(2), Dec. 2016.

[43] Z. Milosevic, M. Biely, and A. Schiper. Bounded delay in Byzantine-tolerant state machine replication. In *Proc. SRDS*, Sept. 2013.

[44] P. Miner. Analysis of the SPIDER fault-tolerance protocols. In *Proc. NASA Langley Formal Methods Workshop (LFM)*, 2000.

[45] G. Neiger. Distributed consensus revisited. *Information processing letters*, 49(4):195–201, 1994.

[46] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 248–262, 1988.

[47] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX ATC*, 2014.

[48] D. A. Osvik. Fast embedded software hashing. Cryptology ePrint Archive, Report 2012/156, 2012. https://eprint.iacr.org/2012/156.

[49] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228234, Apr. 1980.

[50] Rambus. PKA-IP-28 / EIP-28 RSA/ECC Public Key Accelerators. https://www.rambus.com/security/crypto-accelerator-hardware-cores/basic-crypto-blocks/pka-ip-28/.

[51] I. Rouf, R. D. Miller, H. A. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *Proc. USENIX Security*, 2010.

[52] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Proc. NSDI*, 2008.

[53] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. In *Proc. NSDI*, 2009.

[54] T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[55] ST Microelectronics. Introduction to STM32 microcontrollers security. Application note AN5156, available from https://www.st.com/resource/en/application_note/dm00493651-introduction-to-stm32-microcontrollers-security-stmicroelectronics.pdf.

[56] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. P. Bianco, and C. Baisse. Announcing the first SHA1 collision. https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html.

[57] H. Tschofenig and M. Pegourie-Gonnard. Performance of state-of-the-art cryptography on ARM-based microprocessors. NIST Lightweight Cryptography Workshop 2015; available from https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/presentations/session7-vincent.pdf.

[58] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. SOSP*, 2007.

[59] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *Proc. EuroSys*, 2011.

[60] T. Yang, R. Gifford, A. Haeberlen, and L. T. X. Phan. The synchronous data center. In *Proc. HotOS*, May 2019.

[61] G. Yi. Implementing the RSA cryptosystem with the public key accelerator. Analog Devices EE-385, https://www.analog.com/media/en/technical-documentation/application-notes/EE385v01.pdf.

[62] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, 2003.

[63] Y. Zhang and J. Jiang. Bibliographical review on reconfigurable fault-tolerant control systems. *Annual reviews in control*, (32):229–252, 2008.