Distributed Agreement and Its Relation with Error-Correcting Codes*

R. Friedman[1], A. Mostéfaoui[2], S. Rajsbaum[3], and M. Raynal[2]

[1] Department of Computer Science, The Technion, Haifa, Israel,
roy@cs.technion.ac.il,
[2] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France,
{achour,raynal}@irisa.fr,
[3] HP Research Lab, Cambridge, MA 02139, USA and Inst. Matem. UNAM, Mexico,
Sergio.Rajsbaum@hp.com

Abstract. The *condition based* approach identifies sets of input vectors, called *conditions*, for which it is possible to design a protocol solving a distributed problem despite process crashes. This paper investigates three related agreement problems, namely consensus, interactive consistency, and k-set agreement, in the context of the condition-based approach. In consensus, processes have to agree on one of the proposed values; in interactive consistency, they have to agree on the vector of proposed values; in k-set agreement, each process decides on one of the proposed values, and at most k different values can be decided on. For both consensus and interactive consistency, a direct correlation between these problems and error correcting codes is established. In particular, crash failures in distributed agreement problems correspond to erasure failures in error correcting codes, and Byzantine and value domain faults correspond to corruption errors. It is also shown that less restrictive codes can be used to solve k-set agreement, but without a necessity proof, which is still an open problem.

Keywords: Asynchronous Distributed System, Code Theory, Condition, Consensus, Crash Failure, Distributed Computing, Erroneous Value, Error-Correcting Code, Fault-Tolerance, Hamming Distance, Interactive Consistency.

1 Introduction

Context of the paper. Agreement problems are among the most fundamental problems in designing and implementing reliable applications on top of an asynchronous distributed environment prone to failures [3,21]. Among these problems, *consensus* has been the most widely studied. Specifically, in consensus, each process proposes a value, and all processes have to agree on the same proposed value. In the *interactive consistency* problem (initially stated in [30] for the case of Byzantine failures) each process proposes a value, and processes have to agree on the same vector, such that the i-th entry of the vector contains the

* As decided by the program committee, this paper results from the merging of [17] and [24], which were developed separately.

D. Malkhi (Ed.): DISC 2002, LNCS 2508, pp. 63–87, 2002.

value proposed by process p_i if it is correct. Interactive consistency is at least as difficult as consensus: a solution to interactive consistency can be used to solve consensus.

Given the importance of agreement problems in distributed computing, it is remarkable that they cannot be solved in an asynchronous system if only one process can fail, and only by crashing. More precisely, it can be shown that for any protocol that tries to ensure agreement, there is an infinite run in which no process can decide. The first such result is the FLP impossibility for consensus in a message passing system [15], which has been extended to many other agreement problems and distributed models [3,21].

Over the years, researchers have investigated ways of circumventing these impossibility results. The first approach for overcoming the impossibility result considers weaker agreement problems, such as approximate agreement [13], set agreement [10], and randomized solutions [5]. The second approach considers stronger environments that are only partially asynchronous [12,14] and/or have access to failure detectors [11,18]. The third, *condition based* approach, is the focus of this paper [23,25,26,27,34,35]. This approach consists of restricting the set of possible input configurations to a distributed problem. An input configuration can be represented by a vector whose entries are the individual processes' input values in an execution. It has been shown [23] that in asynchronous environments, consensus is solvable despite f failures when the set of allowed input vectors obey certain conditions, which are shown to be necessary and sufficient. This area is also related to the work of [8], which developed an approach for designing algorithms that can utilize some information about the typical conditions that are likely to hold when they are invoked.

The condition based approach can serve two complementary purposes. It can first be viewed as an optimization tool. That is, it allows to identify scenarios where it is always possible to guarantee termination of agreement protocols, yet design the protocols in a way that will prevent them from reaching unsafe decisions even if these scenarios (conditions) where not met. As can be seen from [23, 25,26,27], such scenarios or conditions, are likely to be common in practical systems. Second, as it becomes evident from this work (Section 5), one can utilize the conditions as a guideline to augmenting the environment with optimal levels of synchrony that are cheap enough to support, yet guarantee the solvability of the corresponding agreement problem. In these cases, it is then possible to employ highly efficient protocols that terminate in at most two communication steps even when there are failures. Thus, in comparing this approach to failure detectors, the latter can be viewed as an abstraction that encapsulates the minimal synchrony assumptions on the environment in terms of the ability to detect failures in order to solve agreement problems. On the other hand, a result of this paper shows that the condition based approach can be viewed as capturing minimal requirements on the number of synchronous communication links to solve these problems.

Content of the paper. This paper takes the condition based approach a step further, and establishes a direct relation between error-correcting codes and agree-

ment problems, based on the notion of condition based agreement. In particular, in this paper we obtain the following results. First, we show that the conditions that allow interactive consistency to be solved despite f_c crashes and f_e value domain faults is exactly the set of error correcting codes capable of recovering from f_c erasures and f_e corruptions. Second, we prove that consensus can be solved despite f_c crash failures iff the condition corresponds to a code whose Hamming distance is $f_c + 1$ and Byzantine consensus can be solved despite f_b Byzantine faults iff the Hamming distance of the code is $2f_b + 1$. (There is in fact an additional requirement needed to satisfy the validity property of consensus, as discussed later in the paper.) Third, we show a code that allows solving k-set agreement in both the benign and Byzantine failure models using a simple protocol, but were not able to prove the necessity of the code.

The paper also presents several interesting results that are derived from the main results mentioned above. Namely, we show that by exploring error correcting codes, we can find the parameters needed by our protocols to solve interactive consistency and consensus. On the other hand, coding theory may benefit from this connection. As a simple example, we show that there are no perfect codes that tolerate erasure failures, for otherwise it would have violated the consensus impossibility results. Moreover, we discuss the practical implications of our work to the design choices of distributed consensus algorithms in mixed environments, and introduce the notion of cluster-based failure detectors. Furthermore, our results imply that coding theory can serve as a guideline to efficient deployment and utilization of sparse synchronous communication lines, as promoted, for example, by the *wormholes* approach [36].

The fact that agreement problems are solvable when the input vectors are limited to error correcting codes is not surprising. In particular, error correcting techniques were used in [6] as basic building blocks in constructions for computing arbitrary functions in synchronous systems. Thus, the main challenges of this work include finding algorithms that provide safety always and termination in all favorable circumstances. Additionally, the discussion of the cost of providing agreement, validity, and termination in all possible scenarios shed an important insight into these problems. Finally, the fact that these conditions are necessary for consensus is somewhat surprising, since the problem definition of consensus allows for initial bi-valent configurations. The interesting insight that comes out of the proofs is that these conditions are the minimal requirement to avoid initial bi-valent configurations, and this is exactly why they are required.

Road map. The paper is made up of six sections. Section 2 introduces the computation model and defines the main agreement problems we are interested in (namely, consensus and interactive consistency). Section 3 addresses condition-based interactive consistency and shows that the conditions that allow to solve this problem are exactly error-correcting codes. Section 4 provides a code-based characterization of the conditions that allow to solve consensus. Then, Section 5 focuses on the practical implications of the previous results. Finally, Section 6 concludes the paper. More details and proofs of theorems can be found in [17, 24].

2 Model and Problem Statement

In this paper we assume a standard asynchronous shared memory model or message passing model [3,21] according to our needs, and consider the typical notions of *processes*, *local histories*, *executions*, and *protocols*. We also assume a fixed set of n processes trying to solve an *agreement problem*. In agreement problems, each process has an initial input value, and must decide on an output value. In our model, in each execution at most $f_c < n$ processes can fail by *crashing*. Additionally, up to $f_e < n/2$ processes may suffer *value domain* errors [32] and up to $f_b < n/3$ incur *Byzantine* errors, depending on the circumstances. We sometimes simply write f to denote the total number of failures. A process that suffers a value domain error, also known as *value-faulty*, behaves as if it had a different input value than the one actually given to it, but must otherwise obey the protocol. On the other hand, a process that suffers a Byzantine error, also known as *Byzantine process*, behaves in an arbitrary manner. A process that does not crash and does not suffer any error is called *correct*; otherwise, it is *faulty*.

A universe of values $\mathcal{V}$ is assumed, together with a default value $\perp$ not in $\mathcal{V}$. In the *consensus* problem, each process p_i proposes a value $v_i \in \mathcal{V}$ (the input value of that process), and has to decide on a value (the output value), such that the following properties are satisfied:

- C-Agreement. No two different values are decided.
- C-Termination. A process that does not crash decides.
- C-Validity. A decided value v is a proposed value.

For Byzantine failures, we modify the requirements to be:

- BC-Agreement. No two different values are decided by correct processes.
- BC-Termination. A correct process decides.
- BC-Validity. If all proposed values are the same value v, then the value decided by correct processes is v.

The *interactive consistency* (IC) problem is defined as follows. Each process p_i proposes a value $v_i \in \mathcal{V}$ (the input value), and has to decide a vector D_i (the output value), such that the following properties are satisfied:

- IC-Agreement. No two different vectors are decided.
- IC-Termination. A process that does not crash decides.
- IC-Validity. Any decided vector D is such that $D[i] \in \{v_i, \perp\}$, and is v_i if p_i does not crash.

It is easy to see that, as noted in the Introduction, the IC problem is at least as hard as consensus, and hence unsolvable even if at most one process can crash. Also, it is possible to view the collection of input values to each of these problems as an input vector to the problem. We are interested in conditions on these input vectors that allow the IC and *consensus* problems to be solved despite process failures.

3 Condition-Based Interactive Consistency

3.1 Notation

Let the *input vector* associated with an execution be a vector J, such that $J[i]$ contains the value $v_i \in \mathcal{V}$ proposed by p_i or $\perp$ if p_i crashes initially and does not take any steps. Let $\mathcal{V}^n$ be the set of all possible vectors (of size n) with all entries in $\mathcal{V}$. We typically denote by I a vector in $\mathcal{V}^n$ and by J a vector that may have some entries equal to $\perp$, and hence in $\mathcal{V}^n_{f_c}$, the set of all the n-vectors over $\mathcal{V}$ with at most f_c entries equal to $\perp$. For vectors $J1, J2 \in \mathcal{V}^n_{f_c}$, $J1 \leq J2$ if $\forall k:\ J1[k] \neq \perp \Rightarrow J1[k] = J2[k]$. We define two functions:

- $\#_x(J)$ = number of entries of J whose value is x, with $x \in \mathcal{V} \cup \{\perp\}$.
- $d_{\not\perp}(I, J)$ = number of corresponding non-$\perp$ entries that differ in I and J.

When I has no entry equal to $\perp$, we have $d(I, J) = \#_\perp(J) + d_{\not\perp}(I, J)$. Where $d(I, J)$ is the Hamming distance, i.e., total number of entries where I and J differ. Given a vector $I \in \mathcal{V}^n$,

$$I_{f_c,f_e} \ = \ \{J \mid \#_\perp(J) \leq f_c \ \wedge \ d_{\not\perp}(I, J) \leq f_e \ \}$$

and for a subset C of $\mathcal{V}^n$,

$$\mathcal{C}_{f_c,f_e} \ = \ \bigcup_{I \in C} I_{f_c,f_e}.$$

Thus I_{f_c,f_e} represents a sphere of vectors centered at I and including the vectors J whose distances $\#_\perp(J)$ and $d_{\not\perp}(I, J)$ are bounded by f_c and f_e, respectively. Also, $\mathcal{C}_{f_c,f_e}$ is the union of the spheres centered in vectors of C.

3.2 The CB_IC Problem

As indicated in the Introduction, the idea of the condition-based approach is considering sets of input configurations for which a particular agreement problem can be solved. Such sets C are called conditions. In the IC problem, as in other agreement problems, a condition C is a subset of $\mathcal{V}^n$. It is assumed that C represents input configurations that are common in practice. Hence, it is required that the protocol terminates whenever the input configuration belongs, or could have belonged to C. That is, if the input vector is J (some processes may have crashed initially), termination is required if $J \leq I$ for some $I \in C$, since it is possible that the processes that crashed initially had inputs that would complete J into a vector $I \in C$. Similarly, termination is required if some (at most f_e) non-$\perp$ values of J can be changed to obtain a vector J', such that $J' \leq I$ for $I \in C$, since it is possible that the corresponding processes are value-faulty.

More precisely, here follows a condition-based version of the interactive consistency problem. Each process p_i proposes a value $v_i \in \mathcal{V}$ and has to decide a vector D_i. We say that an (f_c, f_e)-*fault tolerant protocol solves the* CB_IC *problem for a condition* C, if in every execution whose proposed vector J belongs to $\mathcal{V}^n_{f_c}$, the protocol satisfies the following properties:

- CB_IC-Agreement. No two different vectors are decided.
- CB_IC-Termination. If (1) $J \in \mathcal{C}_{f_c,f_e}$ and no more than f_c processes crash, or (2.a) no process crashes, or (2.b) a process decides, then every crash-correct process decides.
- CB_IC-Validity. If $J \in \mathcal{C}_{f_c,f_e}$, then the decided vector D is such that $J \in D_{f_c,f_e}$ with $D \in C$.

The agreement property states that there is a single decision, even if the input vector is not in C, guaranteeing "safety" always. The termination property requires that the processes that do not crash must decide at least when the circumstances are "favorable." Those are (1) when the input could have belonged to C, as explained above, (provided there are no more than f_c crashes during the execution), and (2) under normal operating conditions. The validity property eliminates trivial solutions by relating the decided vector and the proposed vector. It states that, when the proposed vector belongs to at least one sphere defined by the condition, the center of such a sphere is decided, which is one of the possible actual inputs that could have been proposed. To simplify the notation we do not allow $\perp$ entries in the decided vector (the same results apply).

3.3 The Interactive Consistency Conditions

We define here a set of conditions for which there is a solution to the CB_IC problem. Then, the next section presents a protocol that solves CB_IC for any condition in this set. As it can be seen from the definitions below, this set is quite restricted. Nevertheless, in the following section we prove that those are the only conditions for which the CB_IC problem can be solved.

Similarly to the approach used in [23], we define the set of conditions in two equivalent ways, called acceptability and legality. We start with acceptability, which is useful to derive protocols. We then consider legality, which is useful to prove impossibility results.

Acceptability is defined in terms of a predicate P and a function S. Given a condition C and an input vector $J \in \mathcal{V}_{f_c}^n$ proposed by the processes, $P(J)$ has to hold when $J \in \mathcal{C}_{f_c,f_e}$ to allow a process to decide at least in those cases. Then, $S(J)$ provides the process with the corresponding decision vector. To meet these requirements, P and S have to satisfy the following properties.

- Property $\mathrm{T}_{C\to P}$: $I \in C \Rightarrow \ \forall J \in I_{f_c,f_e} : \ P(J)$,
- Property $\mathrm{A}_{P\to S}$:
  $\forall J1, J2 \in \mathcal{V}_{f_c}^n : \ \ (J1 \leq J2) \wedge P(J1) \wedge P(J2) \Rightarrow S(J1) = S(J2)$,
- Property $\mathrm{V}_{P\to S}$:
  $\forall J \in \mathcal{V}_{f_c}^n : \ \ P(J) \ \Rightarrow \ S(J) = I$ such that $I \in C \ \wedge \ J \in I_{f_c,f_e}$.

Definition 1. *A condition C is* (f_c, f_e)-acceptable *if there exist a predicate P and a function S satisfying the three previous properties.*

The following is a more geometric version of acceptability (where $d(,)$ is Hamming distance).

Definition 2. *A condition C is* (f_c, f_e)-legal *if for all distinct $I1, I2 \in C$, $d(I1, I2) \geq 2f_e + f_c + 1$.*

We will prove that the set of (f_c, f_e)-acceptable conditions is the same as the set of (f_c, f_e)-legal conditions, and this is precisely the set of conditions for which there exists an (f_c, f_e)-fault tolerant protocol that solves CB_IC.

3.4 A Shared Memory CB_IC Protocol

We show in this section that for any (f_c, f_e)-acceptable condition C there is a (f_c, f_e)-fault tolerant protocol that solves CB_IC. Such a protocol is described in Figure 1, which needs to be instantiated with parameters P and S associated to C. Interestingly, the protocol is very similar to the condition-based consensus protocol presented in [23], illustrating the relation between consensus and CB_IC.

Computation Model

We consider a standard asynchronous system made up of $n > 1$ processes, $p_1, \ldots, p_n$, that communicate through a single-writer, multi-reader shared memory, and where at most f_c, $1 \leq f_c < n$, processes can crash [3,21].

We assume the shared memory is organized into arrays. The j-th entry of an array $X[1..n]$ can be read by any processes p_i with an operation read$(X[j])$. Only p_i can write to the i-th component, $X[i]$, and it uses the operation write$(v, X[i])$ for this.

To simplify the description of our algorithms, we assume two primitives, collect and snapshot. The collect primitive is a non-atomic operation which can be invoked by any process p_i. It can only be applied to a whole array $X[1..n]$, and is an abbreviation for $\forall j$: **do** read$(X[j])$ **enddo**. Hence, it returns an array of values $[a_1, \ldots, a_n]$ such that a_j is the value returned by read$(X[j])$. The processes can take atomic snapshots of any of the shared arrays: snapshot(X) allows a process p_j to atomically read the content of all the registers of the array X. This assumption is made without loss of generality, since atomic snapshots can be wait-free implemented from single-writer multi-reader registers (although there is a cost in terms of efficiency: the best known simulation has $O(n \log n)$ time complexity [2]).

Each shared register is initialized to a default value $\perp$. In addition to the shared memory, each process has a local memory. The subindex i is used to denote p_i's local variables.

Protocol

A process p_i starts by invoking $SM\_CB\_IC\ (v_i)$ with some $v_i \in \mathcal{V}$. It terminates when it executes the statement **return** (line 7, 9 or 10) which provides it with a

decision vector. The shared memory is made up of two arrays of atomic registers, $V[1..n]$ and $W[1..n]$ (the aim of $V[i]$ is to contain the value proposed by p_i, while the aim of $W[i]$ is to contain the vector p_i suggests to decide on or $\top$ if p_i cannot decide by itself). The protocol has a three part structure.

- Local view determination: lines 1-3. A process p_i first writes into $V[i]$ the value it proposes (line 1). Then, it reads the array V until it gets a vector (V_i) including at least $(n - f_c)$ proposed values.
- Wait-free condition-dependent part: lines 4-5. If $P(V_i)$ holds, p_i computes its view $w_i = [a_1, \ldots, a_n]$ of the decided vector. If p_i cannot decide, it sets w_i to $\top$. p_i also writes w_i in the shared register $W[i]$ to help the other processes decide.
- Termination part: lines 6-11. If $w_i \neq \top$, then p_i unilaterally decides the vector w_i. If it cannot decide by itself ($w_i = \top$) p_i waits until it knows (1) either that another process p_j has unilaterally decided, (2) or that no process can unilaterally decides. In the first case, it decides p_j's suggested vector, while in the second case it decides the full vector of proposed values.

```
Function SM_CB_IC (v_i)

(1)  write(v_i, V[i]);
(2)  repeat V_i ← collect(V) until (#_⊥(V_i) ≤ f_c) endrepeat;
(3)  V_i ← snapshot(V);
(4)  if P(V_i) then w_i ← S(V_i) else w_i ← ⊤ endif;
(5)  write(w_i, W[i]);
(6)  if (w_i ≠ ⊤)
(7)    then return (w_i)
(8)    else repeat W_i ← collect(W)
                 until (∃W_i[j] ≠ ⊥, ⊤) ∨ (W_i = [⊤, ..., ⊤])
           endrepeat;
(9)        if (∃W_i[j] ≠ ⊥, ⊤) then return (W_i[j])
(10)                           else return (collect(V))
(11)       endif
(12) endif
```

Fig. 1. A Shared Memory CB_IC Protocol

Theorem 1. *The protocol described in Figure 1 is an (f_c, f_e)-fault tolerant protocol that solves the* CB_IC *problem for a condition C, when it is instantiated with P, S associated to C, and C is (f_c, f_e)-acceptable.*

Proof Follows directly from the next three Lemmas. $\square_{Theorem\ 1}$

In the following J denotes the vector actually proposed by the processes.

Lemma 1. CB_IC-Validity. *If $J \in \mathcal{C}_{f_c,f_e}$, then the decided vector D is such that $J \in D_{f_c,f_e}$ with $D \in C$.*

Proof There are three cases according to the line (7, 9 or 10) at which a process decides.

- p_i decides at line 7. In that case, $P(V_i)$ held at line 4. Moreover, $V_i \leq J$ and $\#_\perp(V_i) \leq f_c$, from which we conclude that if $J \in \mathcal{C}_{f_c,f_e}$, we also have $V_i \in \mathcal{C}_{f_c,f_e}$. It then follows from $\mathrm{V}_{C\to P}$ that $S(V_i) = I$ with $I \in C$ and $J \in I_{f_c,f_e}$.
- p_i decides at line 9. In that case, p_i decides a vector decided by another process at line 7. CB_IC-Validity follows from the previous item.
- p_i decides at line 10. In that case $W = [\top, \ldots, \top]$, i.e., no process has suggested a decision vector. Hence, for any p_j, $P(V_j)$ was false. Combining this with the $\mathrm{T}_{C\to P}$ property, we get that the input vector J does not belong to $\mathcal{C}_{f_c,f_e}$, and CB_IC-Validity trivially follows.

$\square_{Lemma\ 1}$

Lemma 2. CB_IC-Termination. *If (1) $J \in \mathcal{C}_{f_c,f_e}$ and no more than f_c processes crash, or (2.a) no process crashes, or (2.b) a process decides, then every crash-correct process decides.*

Proof We consider the three cases separately.

- Let us assume that $J \in \mathcal{C}_{f_c,f_e}$ and no more than f_c processes crash.
  Let p_i be a crash-correct process. Let us first observe that, as at most f_c processes crash, p_i cannot block forever at line 2. Moreover, let V_i the local view obtained by p_i. As $J \in \mathcal{C}_{f_c,f_e}$, $V_i \leq J$ and $\#_\perp(V_i) \leq f_c$, we have $V_i \in \mathcal{C}_{f_c,f_e}$, i.e., $\exists D \in C$ such that $V_i \in D_{f_c,f_e}$. It then follows from the $\mathrm{T}_{C\to P}$ property that $P(V_i)$ holds. Consequently, $w_i \neq \top$ and p_i decides at line 7.
- Let us assume that no process crashes and $P(V_i)$ holds for no process p_i. (Hence $J \notin \mathcal{C}_{f_c,f_e}$; otherwise, the previous item would apply.) As there is no crash, no process blocks forever at line 2. Moreover, as $P(V_i)$ holds for no process p_i and no process crashes, W becomes eventually equal to $[\top, \ldots, \top]$. Consequently, no process blocks forever at line 8, and each process executes line 10 and terminates.
- Let us assume that some process (p_j) decides. In that case, as p_j executed line 2, we conclude that at least $(n - f_c)$ processes have deposited the value they propose into V, and consequently, no crash-correct process can block forever at line 2.
  Let us consider the case where p_j decides at line 7. Let p_i be a crash-correct process that does not decide at line 7. As no crash-correct process p_i blocks forever at line 2, p_i benefits from the vector deposited by p_j into $W[j]$ to decide at line 9.
  Let us consider the case where p_j decides at line 9. In that case, some process p_k deposited a vector into $W[k]$. As we have seen just previously, all crash-correct processes decide.

Let us finally consider the case where p_j decides at line 10. Then $W = [\top, \ldots, \top]$, and consequently, no process decided at line 7 or 9. But, as $W = [\top, \ldots, \top]$, any crash-correct process eventually exits line 8 and then decides at line 10.

$\square_{Lemma\ 2}$

The following corollary follows from the proof of the previous theorem.

Corollary 1. *Either all processes that decide do it at lines 7/9, or at line 10.*

Lemma 3. CB_IC-Agreement. *No two different vectors are decided.*

Proof Let us consider two processes p_i and p_j that decide. By Corollary 1, there are two cases. Moreover, if a process decides at line 9, it decides a vector decided by another process at line 7. So, in the following we only consider decisions at line 7 or 10.

- Both p_i and p_j decide at line 7. In that case, their local views V_i and V_j are such that $P(V_i)$ and $P(V_j)$ hold. Since the snapshot invocations can be ordered, these local views are ordered. Without loss of generality let us consider $V_i \leq V_j$. As $V_i \leq V_j$ and both $P(V_i)$ and $P(V_j)$ hold, we conclude from the $A_{P \to S}$ property that $S(V_i) = S(V_j)$.
- Both p_i and p_j decide at line 10. In that case, it follows from the protocol text that they decide the same vector (made up of the n proposed values).

$\square_{Lemma\ 3}$

3.5 Characterizing the Interactive Consistency Conditions

A Characterization
In this section we prove the opposite of Theorem 1: if there is a (f_c, f_e)-fault tolerant protocol that solves CB_IC for C, then C must be (f_c, f_e)-acceptable. The proof extends ideas initially proposed in [9,29] for $f = 1$. We start by establishing a bridge from legality to acceptability.

Lemma 4. *An (f_c, f_e)-legal condition is (f_c, f_e)-acceptable.*

Proof Let C be an (f_c, f_e)-legal condition. We show that there are a predicate P and a function S satisfying the properties $T_{C \to P}$, $A_{P \to S}$ and $V_{P \to S}$ defined in Section 3.3.

As C is (f_c, f_e)-legal, for any two distinct input vectors $I1$ and $I2$ we have $d(I1, I2) \geq 2f_e + f_c + 1$, from which we conclude that $\mathcal{C}_{f_c,f_e}$ is made up of non-intersecting spheres, each centered at a vector I of C. Let us define P and S as follows:

- $P(J)$ holds iff J belongs to a sphere (i.e., $\exists I \in C :\ J \in I_{f_c,f_e}$),
- $S(J)$ outputs the center I of the sphere to which J belongs.

The properties $T_{C \to P}$ and $V_{P \to S}$ are trivially satisfied by these definitions. Let us consider the property $A_{P \to S}$. If $P(J1)$ holds, $J1$ belongs to a sphere centered at $I1 \in C$ (i.e., $J1 \in I1_{f_c,f_e}$). Moreover, due to definition of S, we have $S(J1) = I1$. Similarly for $J2$, if $P(J2)$ holds, $J2$ belongs to a sphere centered at $I2 \in C$ (i.e., $J2 \in I2_{f_c,f_e}$) and $S(J2) = I2$.

If $I1 \neq I2$, we have $d(I1, I2) \geq 2f_e + f_c + 1$ from the legality definition. It follows that there is at least one non-$\perp$ entry in which $J1$ and $J2$ differ. Consequently, in that case, we cannot have $J1 \leq J2$.

Let us now consider the additional assumption stated in $A_{P \to S}$, namely, $J1 \leq J2$. From the previous observation, we conclude that, if $P(J1)$ and $P(J2)$ hold and $J1 \leq J2$, then $J1$ and $J2$ belong to the same sphere, and consequently have the same center I. Hence, $S(J1) = S(J2) = I$. $\square_{Lemma\ 4}$

Lemma 5. *If there is an (f_c, f_e)-fault tolerant protocol that solves* CB_IC *for C, then C must be (f_c, f_e)-legal.* (Proof in [24].)

We can now prove our main result.

Theorem 2. *The* CB_IC *problem for a condition C is (f_c, f_e)-fault tolerant solvable iff C is (f_c, f_e)-legal.*

Proof By Lemma 5 if the CB_IC problem for a condition C is (f_c, f_e)-fault tolerant solvable then C is (f_c, f_e)-legal. By Lemma 4 C is (f_c, f_e)-acceptable. By Theorem 1 the CB_IC problem for C is (f_c, f_e)-fault tolerant solvable. $\square_{Theorem\ 2}$

Correspondence with Error-Correcting Codes

A one-to-one correspondence. Consider an error-correcting code (ECC) problem where a sender wants to reliably send words to a receiver through an unreliable channel. Each word is represented by a sequence of k digits from an alphabet $\mathcal{A}$, and is called a *codeword*. A *code* consists of a set of codewords. The channel can erase or alter digits. The problem is to design a code that allows the receiver to recover the word sent from the word it receives (the received word can contain erased digits[1] and modified digits). We assume all codewords are of the same length, n. The ECC theory has been widely studied and has applications in many diverse branches of mathematics and engineering (see any textbook, e.g., [4]).

Although its goal is different, the CB_IC problem can actually be associated in a one-to-one correspondence with the ECC problem. More precisely, we have the following correspondences:

- alphabet $\mathcal{A}$/set of values $\mathcal{V}$,
- codeword/ vector of the condition,
- codeword length/number of processes (n),

[1] An erasure occurs when the received digit does not belong to the alphabet $\mathcal{A}$. Such a digit is replaced by a default value ($\perp$).

– code/condition,
– erasure/process crash (upper bounded by f_c),
– alteration/erroneous proposal (upper bounded by f_e),
– word received/proposed input vector,
– decoding/deciding.

On the necessary and sufficient condition. We have stated in Theorem 2 that the CB_IC problem is (f_c, f_e)-fault tolerant solvable for a condition C iff $\forall I1, I2 \in C$: $(I1 \neq I2) \Rightarrow d(I1, I2) \geq 2f_e + f_c + 1$. The "corresponding" code theory theorem (whose proof appears in any texbook on error-correcting codes, e.g., Theorem 5.17 in [4], pages 96-97) is stated as follows:

> "A code C is t *error/*e *erasure* decoding iff its minimal Hamming distance is $\geq 2t + e + 1$."

In this sense, CB_IC and ECC are equivalent problems:

Theorem 3. *The* CB_IC *problem is* (f_c, f_e)*-fault tolerant solvable for a condition* C *iff* C *is* f_e error/f_c erasure *decoding.*

4 A Coding Theory-Based Approach to Consensus

An input vector can actually be seen as a codeword made up of n digits (one per process) that encode the decision value. This observation [17] leads to another way to characterize the set of conditions that allow to solve the consensus problem. The discussion that follows uses the message passing model. As shown in [17], the results hold also in the shared memory model. So, similarly to [23], the initial input values of all processes is seen as a vector of $\mathcal{V}^n$ (the set of all possible input vectors). We say that an f*-fault tolerant protocol solves the consensus problem for a given condition* C if it guarantees the following properties[2]:

– CB_C-Validity. If the input vector belongs to the condition C, then a decided value is a proposed value.
– CB_C-Agreement. If the input vector belongs to the condition C, then no two different values are decided.
– CB_C-Guaranteed Termination. If at most f processes fail and the input vector belong to C, then every correct process p_i eventually decides.

For Byzantine failures we use the following definition of validity and agreement:

– CB_BC-Validity. If all input values are v, then only v can be decided by a correct process.
– CB_BC-Agreement. If the input vector belongs to the condition C, then no two different values are decided by correct processes.

[2] Notice that this condition-based definition of the consensus problem is weaker than the one that we introduced in [23]. The definition in [23] requires validity and agreement to hold even when the input vector does not belong to the condition, (i.e., C-Validity and C-Agreement), and termination to additionally hold whenever there are no failures or a process decides. Section 4.2 discusses these issues.

4.1 Characterizing the Input Vectors with Codes

We define the initial configuration c of the system as *bi-valent* if there are two executions σ_1 and σ_2 that start with c such that the decision value in σ_1 is v_1, the decision value in σ_2 is v_2, and $v_1 \neq v_2$. Otherwise, the configuration is *univalent*.

In our characterization, each allowed input vector must always lead the system to the same decision value. In other words, the initial configuration of the system is not allowed to be bi-valent. Thus, we can treat the set of allowed input vectors as codewords coding the possible decision values. Clearly, in the case of consensus, since the decision value has to be unique, the code maps words from $\mathcal{V}^n$ to values in $\mathcal{V}$. Also, due to the validity requirement of consensus, we must limit ourselves to codes in which at least one of the digits of every codeword corresponding to a value $v \in \mathcal{V}$ has to be v. Thus we have:

Definition 3. *A* d-admissible code *is a mapping* $C : \mathcal{V}^n \rightarrow \mathcal{V}$ *such that the Hamming distance of every two codewords coding different values in* $\mathcal{V}$ *is at least* d *and at least one of the digits in each codeword mapped to a value* $v \in \mathcal{V}$ *is* v.

Solving Consensus with d-Admissible Codes

A simple generic protocol for solving the consensus problem using d-admissible codes in both the crash failure and Byzantine failure models is presented in Figure 2 (this protocol combines protocols described in [17,23]). V_i is the vector of initial values heard so far by process p_i; *code* is a set of codewords defining the allowed input vectors.

Function *MP_Consensus*(v_i)

(1) $\forall j$: send VAL1(v_i, i) to p_j;
(2) **wait until** (at least $(n - f)$ VAL1 msgs have been delivered);
(3) $\forall j$: **do if** VAL1(v_j, j) has been delivered **then** $V_i[j] \leftarrow v_j$
(4) **else** $V_i[j] \leftarrow \perp$ **endif**
(5) **enddo**;
(6) $w_i \leftarrow$ match $(V_i$,*code*$)$;
(7) **if** $w_i \neq \perp$ **then return**(w) **endif**

Fig. 2. A Message-Passing Consensus Protocol

For the crash failures model with at most $f = f_c$ failures, it is sufficient to use an $(f_c + 1)$-admissible code, while for the Byzantine failures model with at most $f = f_b$ failures we have to use a $(2f_b + 1)$-admissible code. Also, the protocol uses a subroutine called `match` to check whether the digits received so far can be matched to any codeword. It returns $\perp$ if no matching was found; otherwise, it returns the value of the decoded word corresponding to the codeword matched.

More precisely, each digit that was not received from some process is treated as the special value $\perp$ in the vector of received digits. Then, all `match` has to do is to check if the distance of this vector is at most f from some legal codeword. If it is, `match` returns the value mapped to by this codeword. Otherwise, `match` returns $\perp$. Note that by the specific codes that we use, the same rule applies in both benign and Byzantine failures, and in both cases, if there are at most f failures, we are guaranteed to find such a codeword. Also, it is possible that there would be several possible codewords to chose from, but in this case they all have to be mapped to the same value, so any of them can be picked. The `match` routine is the equivalent of the predicate P and function S introduced in [23]. The exact implementation of the `match` routine is outside the scope of this paper. Here we simply rely on coding theory to guarantee that it exists.

It is not hard to show [17] that the previous protocol satisfies validity, agreement, and termination when the input vectors are indeed codewords of some $(f_c + 1)$-admissible (or $(2f_b + 1)$-admissible) code. Section 4.2 discusses the implications of satisfying validity and agreement when the input vectors are not always codewords, and termination when the input vectors are not codewords but there are no failures.

Necessity of d-Admissibility for Solving Consensus

The consensus problem considered in Theorem 4 (resp. Theorem 5) is defined by the following three properties: CB_C-Agreement (resp.CB_BC-Agreement), CB_C-Guaranteed Termination and CB_C-Validity (resp. CB_BC-Validity).

Theorem 4. *If a condition C allows to solve consensus in the crash failure model (with at most f_c failures), then C consists of codewords of an $(f_c + 1)$-admissible code.*

Proof To prove the theorem, we first point the reader to the proof of the consensus impossibility result as stated in [3]. In that proof, it was shown that if there is a bi-valent initial configuration, then consensus cannot be solved. That proof is for the case of a single failure, but the case of f_c failures is completely analogous. So, all that we have to show is that if the initial input vectors allowed by the condition are not words of an $(f_c + 1)$-admissible code, then there has to be a bi-valent initial configuration.

Assume, by way of contradiction, that C does not correspond to an $(f_c + 1)$-admissible code and there is no bi-valent initial configuration. Thus, there are two allowed univalent initial configurations c_1 and c_2 that differ in less than $f_c + 1$ processes, yet each one leads to a different value v_1 and v_2 respectively. Denote by $P' = p_{i_1}, \ldots, p_{i_k}$ $(k \leq f_c)$ the processes that differ between c_1 and c_2. Hence, there is an execution σ_1 of the protocol that starts at c_1 in which all processes in P' fail before managing to take any action. All other processes must decide in σ_1 on value v_1 without receiving any message from any process in P'. Let p_j be one of the processes that decides in σ_1 on v_1, HP_{p_j} be the history

prefix of p_j at the moment it decides, and $CH_{p_j}(\sigma_1, HP_{p_j})$ be its causal history at that point.

Since the network latency is unbounded, we can create another execution σ_2 that starts in c_2, no process fails during σ_2, but $CH_{p_j}(\sigma_1, HP_{p_j})$ is also a causal history of p_j in σ_2. Given the determinism of p_j, it must also decide in σ_2 on the same value v_1. Since the protocol is assumed to solve consensus, all processes must decide v_1. Thus, either c_2 leads to v_1, or c_2 is bi-valent. A contradiction. $\square_{Theorem\ 4}$

Theorem 5. *If a condition C allows to solve consensus in the Byzantine failure model (with at most f_b failures), then C consists of codewords of a $(2f_b + 1)$-admissible code.*

Proof To prove the theorem, we first note that the proof in [3] that consensus is not solvable if there is an initial bi-valent configuration is also valid for the Byzantine case. Thus, all that we have to show is that if the initial input vectors allowed by the condition are not words of a $(2f_b+1)$-admissible code, then there has to be a bi-valent initial configuration.

Assume by way of contradiction that C does not correspond to a $(2f_b + 1)$-admissible code and there is no bi-valent initial configuration. Thus, there are two allowed univalent initial configurations c_1 and c_2 that differ in less than $2f_b + 1$ processes, yet each one leads to a different value v_1 and v_2 respectively. We can divide the set of processes whose initial state is different in c_1 and c_2 into two subsets, P' and P'' such that $|P'| \leq |P''| \leq f$. Due to termination, there is an execution σ_1 of the protocol that starts at c_1 in which all processes in P' crash before managing to take any action. All other processes must decide in σ_1 on value v_1 without receiving any message from any process in P'. Let p_j be one of the processes in $N \setminus P''$ that decides in σ_1 on v_1, HP_{p_j} be the history prefix of p_j at the moment it decides, and $CH_{p_j}(\sigma_1, HP_{p_j})$ be its causal history at that point.

Since the network latency is unbounded, we can create the following execution σ_2 that starts in c_2. In σ_2, all processes in P'' suffer the Byzantine failure that makes them behave as if their initial configuration was as in c_1, but otherwise obey the protocol. No process crashes during σ_2. Due to the unbounded message latency, all messages of processes in P' are delayed enough so that $CH_{p_j}(\sigma_1, HP_{p_j})$ is also a causal history of p_j in σ_2. Given the determinism of p_j, it must also decide in σ_2 on the same value v_1. Since the protocol is assumed to solve consensus, all correct processes must decide v_1. Thus, either c_2 leads to v_1, or c_2 is bi-valent. A contradiction. $\square_{Theorem\ 5}$

4.2 Strict Condition-Based Consensus

In order to weaken the dependency of the possible solutions on whether the input vectors are indeed codewords, we can make any of the validity, agreement, and termination requirements more strict. Here we discuss the consequences of doing so.

Agreement

In order to ensure that agreement holds even when the input vectors are not codewords (i.e., C-Agreement), it is possible to augment the protocol described in Figure 2 with additional checks that verify that all decided values are the same. The modified protocol is depicted in Figure 3. It guarantees agreement whenever $f = f_c < n/2$ in the benign failures model and when $f = f_b < n/3$ in the Byzantine model. The protocol also guarantees CB-C-Validity and CB-C-Guaranteed Termination.

```
Function MP_Consensus(v_i)

(1)  ∀j : send VAL1(v_i, i) to p_j;
(2)  wait until (at least (n − f) VAL1 msgs have been delivered);
(3)  ∀j : do if VAL1(v_j, j) has been delivered then V_i[j] ← v_j
(4)                                              else V_i[j] ← ⊥ endif
(5)       enddo;
(6)  w_i ← match (V_i,code);
(7)  ∀j : send VAL2(w_i, i) to p_j;
(8)  repeat  wait for a new VAL2(w_j, j) message;
(9)          if VAL2(w, −) with the same non-⊥ value w has
(10)           been received from at least (n − f) processes
(11)           then return(w)
(12)         endif
(13) endrepeat
```

Fig. 3. An Always Safe Message-Passing Consensus Protocol

Validity

The previous protocol (Figure 3) might not satisfy the C-Validity property when the input vector is not a codeword of an admissible code (it only satisfies CB_C-Validity). A slightly stronger definition of admissibility allows to overcome this problem, namely:

Definition 4. *A* strongly d-admissible code *is a mapping* $C : \mathcal{V}^n \to \mathcal{V}$ *such that the Hamming distance of every two codewords coding different values in* $\mathcal{V}$ *is at least* d *and at least* d *of the digits in each codeword mapped to a value* $v \in \mathcal{V}$ *are* v.

We say that an f*-fault tolerant protocol solves the* strict *consensus problem for a given condition* C if it guarantees C-Validity, C-Agreement, and CB-C-Guaranteed Termination. Clearly, the protocol of Figure 3 solves strict consensus for strongly $(f + 1)$-admissible codes.

Theorem 6. *A condition C allows to solve* strict *consensus in the crash failure model (with at most f_c failures), iff C consists of codewords of a* strongly (f_c+1)*-admissible code.*

Proof First, note that the proof of Theorem 4 holds here as well. The only thing we need to show is that C-Validity cannot be guaranteed unless the code satisfies the property that each word is mapped to a value that appears in at least $f_c + 1$ of its digits. Assume by way of contradiction that there is a protocol $\mathcal{P}$ that solves strict consensus for a condition C that includes an allowed input vector V in which no value appears more than $k \leq f_c$ times. In other words, every execution of $\mathcal{P}$ that starts with V has to terminate.

As discussed before, since $\mathcal{P}$ solves consensus for C, V must be an initial univalent configuration. Consider an execution of $\mathcal{P}$ that starts with V and decides some value v. Thus, every execution of $\mathcal{P}$ that starts with V must decide v. Since v only appears $k \leq f_c$ times in V, the execution E in which all processes whose initial value is v crash before sending any message must also terminate with a decision value v. Denote the set of corresponding processes by S. Therefore, there exists an execution E' in which the input vector is the same as V except for all processes in S for which the input value is different, and during E' all processes in S crash immediately. For processes outside S, E' is indistinguishable from E, and therefor E' also terminates with a decision value v. However, this violates C-Validity. $\square_{Theorem\ 6}$

Note that instead of using strongly d-admissible codes, we could use a weaker definition of validity that only requires it to hold if either all initial values are the same, or when there are no failures. Such a definition is used in any case for the Byzantine failure model.

It is shown in [17] that strongly $(f + 1)$-admissible codes are the same as f-acceptable codes in [23]. In particular, Condition C_1 in [23] requires that the most popular value in a vector in C_1 appear at least $f + 1$ times more than the second most popular value in the same vector. Clearly, any two vectors that lead to different decision values and obey this condition must differ in at least $f + 1$ places, and thus the Hamming distance of the code is $f + 1$.

Condition C_2 in [23] requires that the largest value in a vector in C_2 appear at least $f + 1$ times. For any condition defined on values from some range $\mathcal{V}$, any vector mapped to a value $v \in \mathcal{V}$ must include at least $f + 1$ v entries and no entries larger than v. Consider two vectors V_1 and V_2 leading to different decision values v and u such that $v > u$. Thus, V_1 must include at least $f + 1$ v entries, while V_2 does not include any v entries, which means that they differ in at least $f + 1$ entries. In other words, the Hamming distance of the code is $f + 1$.

Termination

To guarantee termination in the crash failure model with $f_c < n/2$, both (1) when the input vector is a codeword and (2) when the input vector is not a codeword and there are no failures or a process decides, it is possible to use the

protocols described in [23]. At present, we have no solution for the Byzantine model.

4.3 *k*-Set Consensus

The augmented definition of k-set consensus when there are conditions on the input vectors is:

- CB_K-Validity. If the input vector belongs to the condition C, then a decided value is a proposed value.
- CB_K-Agreement. If the input vector belongs to the condition C, then at most k distinct values are decided.
- CB_K-Guaranteed Termination. If at most f processes fail and the input vector belongs to the condition C, then eventually every correct process p_i decides.

For the case of Byzantine faults we use the following definitions of validity and agreement:

- CB_KB-Validity. If the initial value of all processes is the same, then every correct process that decides has to decide on this value.
- CB_KB-Agreement. If the input vector belongs to the condition C, then at most k distinct values are decided by the correct processes.

Next, we define (d, k)-admissible codes:

Definition 5. *A (d, k)-admissible code is a mapping $C : \mathcal{V}^n \rightarrow \mathcal{V}$ such that: (a) for every codeword w in C, all codewords whose Hamming distance from w is less than d are mapped to at most k different values, and (b) at least one of the digits in each codeword mapped to a value $v \in \mathcal{V}$ is v.*

Given the above definition, we claim that k-set consensus can be solved if the input vectors are codewords of a $(f_c + 1, k)$-admissible code in the crash failures model, and $(2f_b + 1, k)$-admissible code in the Byzantine model. Note that by slightly changing the behavior of the `match` routine in the protocol in Figure 2, we can use the protocol without any additional modifications to solve k-set agreement for the above codes. The only difference is that now `match` checks whether the Hamming distance of the vector of received digits from any codeword is at most f_c (f_b). If the answer is yes, `match` returns the value pointed to by the closest of these codewords, breaking symmetry arbitrarily. Otherwise, `match` returns $\perp$.

5 Practical Implications of the Results

5.1 Benefiting from Error-Correcting Code Theory

Following the results of Sections 3 and 4, we can use known *t error/e erasure* correcting codes to define conditions suited to the CB_IC and *consensus* problems. This is illustrated below with a simple example using a linear code. Interestingly, linear codes can be composed. An additional benefit of using coding theory is that some codes have been proved to be maximal. Using such a code gives a condition that contain as many input vectors as possible.

From a code ... Let us consider the following linear code C, namely, the binary $[n, M, d]$-code where $n = 6$ is the length of a codeword, $M = 2^k = 8$ is the number of codewords (k being the length of the words to be encoded) and $d = 3$ is the code minimal Hamming distance. As $d = 3$, C can detect and correct one error. The set C of codewords can be obtained from a generator matrix G made up of k linearly independent size n vectors. Considering a particular generator matrix G, we get the following set C of 8 codewords:

0 0 0 0 0 0	1 1 1 0 0 0	0 1 0 1 0 1	1 0 1 1 0 1
1 0 0 1 1 0	0 1 1 1 1 0	1 1 0 0 1 1	0 0 1 0 1 1

These codewords can also be defined from a check matrix A. This matrix is obtained from G: it is such that $A\ G^T = 0$ (G^T is the transpose of G, and 0 is a $k \times k$ zero matrix). We have:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Let w be a received word. It is a codeword if its syndrome is equal to the 0 vector (the syndrome of a vector w is the value Aw^T, where w^T is the transpose of w). When $A\ w^T \neq 0$, the syndrome value defines the altered position of w.

... To a condition. Let us now consider a system made up of $n = 6$ processes. The previous 8 vectors define a condition C on the vocabulary $\mathcal{V} = \{0, 1\}$. Due to Definition 2 and Theorem 2, C can cope with $f_c = 2$ process crashes and $f_e = 0$ erroneous proposals (or alternatively, $f_c = 0$ and $f_e = 1$). The matrix A provides a simple way to define the condition C for the CB_IC problem:

$$C = \{I \text{ such that } \mathit{syndrome}(I) = 0\}$$

where $\mathit{syndrome}(I) = A\ I^T$. It is possible to show that the following acceptability parameters (predicate P and function S) can be associated with such a condition C:

$$P(J) = \exists I : \text{ such that } J \in I_{f_c,f_e} \ \wedge\ \mathit{syndrome}(I) = 0,$$
$$S(J) = I \text{ such that } J \in I_{f_c,f_e} \ \wedge\ \mathit{syndrome}(I) = 0.$$

The use of the *syndrome* function allows for an efficient determination of the input vectors I defining a legal condition. As the CB_IC problem directly considers input vectors of size n (each corresponding to a codeword), the generator matrix G is useless for this problem.

5.2 Benefiting from Distributed Computing Results

Let a code C, $|C| > 1$, be (f_c, f_e)-*perfect* if the spheres whose centers are the vectors I of C define a partition of $\mathcal{V}^n_{f_c}$, i.e.,

$$\forall I1, I2 \in C : I1 \neq I2 : \; I1_{f_c,f_e} \cap I1_{f_c,f_e} \; = \; \emptyset \qquad \text{and} \qquad \mathcal{V}^n_{f_c} \; = \; \bigcup_{I \in C} I_{f_c,f_e}.$$

When $f_c = 0$, the notion of (f_c, f_e)-*perfect* code is the coding theory notion of *perfect* code. Interestingly, perfect codes are known. (They are rare, and have the property not to correctly decode a received word with more than f_e errors.) The following is a simple example of using distributed computing to prove a result in coding theory. It states that code perfection does not exist when a code system has to cope with digit erasures.

Theorem 7. *There is no (f_c, f_e)-perfect code for $f_c \geq 1$, $f_e < n$.*

Proof Let us assume that such an (f_c, f_e)-perfect code C does exist. Then, due to the definition of (f_c, f_e)-perfection, we have $\mathcal{V}^n_{f_c} = \bigcup_{I \in C} I_{f_c,f_e}$, i.e., all the possible input vectors are included. As C is (f_c, f_e)-perfect, no two spheres intersect. Consequently, C is (f_c, f_e)-legal, i.e., (f_c, f_e)-acceptable. This means that there is a CB_IC protocol that always terminates when the number of crashes is $\leq f_c$. Consequently, in every execution all decided vectors are equal. Moreover, at least two different vectors are decided, since the code is non-trivial. This is a version of consensus that is known to be unsolvable in the presence of crashes (an implication of FLP [15], e.g. [22]). It follows that C is not $(f_c, 0)$-perfect. $\square_{Theorem\ 7}$

Another trivial result that stems from this work is that error correcting data, e.g., parity bits (and in general `xor`), cannot be computed in asynchronous environments prone to failures.

5.3 Agreement in Mixed Environments

The practical implication of the previous coding-based characterization is the ability to solve consensus in mixed environments. That is, assume that a set of n processes is split into clusters, where in each cluster the communication is synchronous enough so that consensus can be solved, but between clusters the system is asynchronous. Clusters can correspond to different LANs, or a single large LAN can be arbitrarily divided into several clusters for scalability purposes. With such a division into clusters, it is possible to have all nodes of a single cluster initially decide on one value, and use that value as their input value to the global consensus problem, which will be run among all processes using the protocol similar to the one described in Figure 2. If the size of the smallest cluster is at least $f_c + 1$ (resp., $2f_b + 1$), we can solve consensus in the global system despite f_c crash failures (resp., f_b Byzantine failures).

A shortcoming of this approach is that if clusters become disconnected, the previous protocol can block until they reconnect again. Another shortcoming of the above scheme is that it requires a high degree of redundancy in the system. However, if we look at error correcting codes, for both erasure and alteration errors, there are more efficient codes. For example, parity can be used to overcome one digit erasure with only one extra digit, while Hamming code can correct

one digit flip with an overhead of $\log(n)$ digits. But, in both cases, some digits in each codeword depend on many other digits in the same word. Practically speaking, given a code, if some digit b depends on the values of some other digits $b_1, \ldots, b_l$, it indicates that process b needs synchronous communication links with $b_1, \ldots, b_l$. Thus, it would be interesting to find codes that present a good tradeoff between the number of digits each digit depends on, and the digit overhead for error correction. That is, small overhead implies the ability to solve consensus with small hardware redundancy, while low dependency between digits means that it can be applied more easily to real settings, since it requires weaker synchrony assumptions. Looking at linear codes might be a good direction for this [4,7].

Let us notice that Pfitzmann and Waidner have shown how to solve Byzantine agreement for any number of faults in the presence of a reliable and secure multicast during a precomputation phase [31]. Also, Fitzi and Maurer showed how to obtain Global Broadcast in the presence of up to $n/2$ Byzantine failures based on a Local Broadcast service [16]. However, none of these works draws any relation from agreement to error correction.

5.4 Cluster-Based Failure Detectors

The above discussion indicates that it is possible to solve consensus despite a *small number of failures* using failure detectors that provide the accuracy and completeness properties of $\diamond\mathcal{W}$ only among members of clusters. Such failure detectors need not guarantee anything about failure suspicions of processes outside the cluster. Formally, we assume that processes are divided into non-overlapping clusters, and augment the definitions of accuracy and completeness given in [11] as follows:

- Strong c-Completeness. Eventually, every process that fails is permanently suspected by every non-faulty process in the same cluster.
- Weak c-Completeness. Eventually, every failed process is permanently suspected by some non-faulty process in the same cluster.
- Eventual Strong c-Accuracy. There is a time after which no non-faulty process is suspected by any non-faulty process in the same cluster.
- Eventual Weak c-Accuracy. There is a time after which some non-faulty process is not suspected by any non-faulty process in the same cluster.

As discussed in [11], guaranteeing one of these properties is trivial. The difficult problem (impossible in completely asynchronous systems) is guaranteeing a combination of one of the accuracy requirements with one of the completeness requirements. A failure detector belongs to the class c-$\diamond\mathcal{W}$ if it guarantees Weak c-Completeness and Eventual Weak c-Accuracy. Similarly, a failure detector belongs to the class c-$\diamond\mathcal{S}$ if it guarantees Strong c-Completeness and Eventual Weak c-Accuracy.

Clearly, it is possible to simulate a failure detector in c-$\diamond\mathcal{S}$ from a failure detector in c-$\diamond\mathcal{W}$ by running within each cluster the simulation given in [11] for

simulating $\diamond\mathcal{S}$ from $\diamond\mathcal{W}$. It is thus possible to solve consensus among members of the same cluster using c-$\diamond\mathcal{S}$ and any of the $\diamond\mathcal{S}$-based consensus protocols (e.g., [11,19,20,28,33]). Similarly to the discussion above, each process can use the decision value of its cluster as its input value in the global consensus protocol of Figure 2. We call this the *direct cluster based approach*.

On the other hand, it is easy to derive a failure detector in $\diamond\mathcal{W}$ from a failure detector in c-$\diamond\mathcal{W}$. Specifically, assume that each process is equipped with a failure detector FD from the class c-$\diamond\mathcal{W}$.

Theorem 8. *A failure detector FD' that adopts the failure suspicions of FD for processes inside the cluster, but never suspects any process outside the cluster is in $\diamond\mathcal{W}$.*

Proof Note that $FD' \in c$-$\diamond\mathcal{W}$, since it behaves the same as FD for processes inside the same cluster. Clearly, Weak c-Completeness is stronger than Weak Completeness. This is because the latter only requires that eventually, every failed process be permanently suspected by some non-faulty process, but different failed processes can be suspected by different non-faulty processes.

Also, for each cluster, FD guarantees that there it at least one non-faulty process that is not suspected by any non-faulty process within the cluster. Moreover, by the construction of FD', this process is not suspected by any process outside the cluster, and thus FD' is in $\diamond\mathcal{W}$. $\square_{Theorem\ 8}$

Consequently, it is possible to employ Theorem 8 to simulate a failure detector in $\diamond\mathcal{W}$, and use it to solve consensus with any of the previously cited protocols. However, we argue that the direct cluster based approach is more efficient and scalable. That is, the direct cluster based approach only requires failure detection (heartbeats) among nodes of the same cluster. Specifically, there is no need for long haul failure detection, and heartbeats are exchanged only among a small set of close nodes. In contrast, the simulation of $\diamond\mathcal{S}$ from $\diamond\mathcal{W}$ given [11] requires many long haul message exchanges. Moreover, with the direct cluster based approach, all rounds of $\diamond\mathcal{W}$-based protocols are executed between a small set of well connected processes. Given that consensus can be used as a building block for solving other problems in distributed computing this can serve as a basis for a scalable solution to these problems as well.

As before, the downside of this scheme is that if a single cluster becomes disconnected from the rest of the network, this might prevent the global consensus from terminating until that cluster reconnects. Conversely, existing protocols for solving consensus (with respect to the entire set of nodes) that rely on $\diamond\mathcal{S}$ can overcome up to $\lceil n/2 \rceil - 1$ crash failures.

6 Discussion

This paper has investigated principles that underlie distributed computing by establishing links between agreement problems and error correcting codes. The results that have been presented draw a correlation between crash failures in distributed computing and erasures in error correcting, and between value domain

faults and Byzantine failures in distributed computing and digits corruptions in error correcting. In particular, it has been shown that condition-based interactive consistency and error correcting are two facets of the same problem. Similar conditions are shown to be sufficient and necessary for solving consensus. Yet, the conditions for consensus are on one hand less restrictive, since in consensus each decision value may be coded by more than one codeword. On the other hand, the conditions for consensus include a restriction on the occurrence of the decoded value in the digits of the corresponding codewords, which is due to the validity requirement of consensus. (This requirement is implicit in interactive consistency.) For k-set agreement, we only showed sufficient conditions, which are less restrictive due to the more relaxed agreement requirement of the problem. Showing necessary error-correcting based conditions for k-set agreement is still an open issue, although some advancements have been made in giving topological characterizations for such conditions [1,27]. Interestingly enough, the same protocol, instantiated with the appropriate conditions, can be used to solve all three problems [27].

The paper has also discussed some interesting tradeoffs related to whether the agreement, validity, and termination requirements should be preserved only when the input belongs to the condition. Specifically, we showed that in order to always obtain validity (i.e., even when the input vector does not belong to the condition), it is necessary to use more restrictive conditions, while to obtain always agreement and also termination when there are no failures, we need to enrich the basic protocols. We have also discussed the implications of our results in terms of applying limited synchrony technologies like wormholes [36] and cluster based failure detectors in a clever way.

Another interesting result that comes from this paper is that interactive consistency is harder than consensus in the sense that it requires more restrictive conditions. This echos the known result that interactive consistency requires perfect failure detectors [18], while consensus can be solved with unreliable failure detectors (this suggests that it might not be a good idea to solve consensus via interactive consistency). It would be interesting to find a more direct linkage between the strength of conditions and failure detectors. In particular, if one can unify this with topological characterization of such conditions, it might enable viewing failure detectors as topological tweaks on the environment that enable agreement to be solved.

Finding a linkage between coding theory and agreement problems in distributed computing seems to be both an intellectually challenging task and a practically relevant aim. Coding theory is an area that has been extensively studied. By applying results from coding theory, it might be possible to find simpler proofs to existing results, and ideally, even to obtain new results in distributed computing. At the same time, the paper has shown that it is possible to draw simple proofs for results in error correcting codes by reduction to agreement problems.

Our work leaves a few additional interesting open problems. For example, can the results be generalized under a unified framework for general agreement

problems? Is there a distributed computing problem that is a counterpart of error detection codes similarly to the fact that CB_IC is the counterpart of error correction? More generally, given a condition C, which agreement problem does C allow to solve? And at what cost (see [25] for a related result in the case of consensus)?

References

1. Attiya H. and Avidor Z., Wait-Free n-Consensus When Inputs are Restricted. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, These proceedings.
2. Attiya H. and Rachman O., Atomic Snapshots in $O(n \log n)$ Operations. *SIAM Journal on Computing,* 27(2):319–340, 1998.
3. Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* McGraw-Hill, 451 pages, 1998.
4. Baylis J., *Error-Correcting Codes: a Mathematical Introduction*. Chapman & Hall Mathematics, 219 pages, 1998.
5. Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, pp. 27–30, Montréal, 1983.
6. Ben-Or M., Goldwasser S., and Wigderson A., Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *Proc. 20th ACM Symposium on Theory of Computing (STOC'88)*, pp. 1–10, 1988.
7. Berlekamp E.R., *Algebraic Coding Theory* (Second Edition). Aegean Park Press, 1984.
8. Berman P. and Garay J., Adaptability and the Usefulness of Hints. *6th European Symposium on Algorithms (ESA'98),* Springer-Verlag LNCS #1461, pp. 271–282, Venice (Italy), 1998.
9. Biran O., Moran S. and Zaks S., A Combinatorial Characterization of the Distributed 1-Solvable Tasks. *Journal of Algorithms*, 11:420–440, 1990.
10. Chaudhuri S., More *Choices* Allow More *Faults:* set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.
11. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.
12. Dolev D., Dwork C. and Stockmeyer L., On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, 1987.
13. Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E., Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM*, 33(3):499–516, 1986.
14. Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
15. Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.
16. Fitzi M. and Maurer U., From Partial Consistency to Global Broadcast. *Proc. 32rd ACM Symposium on Theory of Computing (STOC'00)*, pp. 494–503, 2000.
17. Friedman R., A Simple Coding Theory-Based Characterization of Conditions for Solving Consensus. *Technical Report CS-2002-06 (Revised Version)*, Department of Computer Science, The Technion, Haifa, Israel, June 30, 2002.
18. Hélary J.-M., Hurfin M., Mostéfaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. *IEEE Trans. on Parallel and Distributed Systems*, 11(9):897–910, 2000.

19. Hurfin M., Mostefaoui A. and Raynal M., A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395–408, 2002.
20. Hurfin M. and Raynal M., A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. *Distributed Computing*, 12(4):209–223, 1999.
21. Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.
22. Moses Y. and Rajsbaum S., The Unified Structure of Consensus: a Layered Analysis Approach. *Proc. 17th ACM Symp. on Principles of Distributed Computing (PODC'98)*, pp. 123–132, 1998. To appear *SIAM Journal on Computing.*
23. Mostefaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Proc. 33rd ACM Symposium on Theory of Computing (STOC'01)*, pp. 153–162, Crete (Greece), 2001.
24. Mostefaoui A., Rajsbaum S. and Raynal M., Asynchronous Interactive Consistency and its Relation with Error-Correcting Codes. *Research Report #1455*, IRISA, University of Rennes, France, April 2002, 16 pages. (Also Brief Announcement in *PODC'02.*) http://www.irisa.fr/bibli/publi/pi/2002/1455/1455.html.
25. Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., A Hierarchy of Conditions for Consensus Solvability. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press pp. 151–160, Newport (RI), 2001.
26. Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Efficient Condition-Based Consensus. *8th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO'01)*, Carleton Univ. Press, pp. 275–291, Val de Nuria (Catalonia, Spain), 2001.
27. Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Condition-Based Protocols for Set Agreement Problems. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, These proceedings.
28. Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int. Symposium on Distributed Computing (DISC'99)*, Bratislava (Slovaquia), Springer-Verlag LNCS 1693, pp. 49–63, 1999.
29. Moran S. and Wolfstahl Y., Extended Impossibility Results for Asynchronous Complete Networks. *Information Processing Letters,* 26:145–151, 1987.
30. Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.
31. Pfitzmann B. and Waidner M., Unconditional Byzantine Agreement for any Number of Faulty Processors. *Proc. of the 9th Int. Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag LNCS 577, pp. 339–350, 1992.
32. Powell D., Failures Mode Assumptions and Assumption Coverage. *Proc. 22th IEEE Fault-Tolerant Computing Symposium (FTCS'92)*, IEEE Society Press, pp. 386–395, Boston (MA), 1992.
33. Schiper A., Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:194–157, 1997.
34. Taubenfeld G., Katz S. and Moran S., Impossibility Results in the Presence of Multiple Faulty Processes. *Information and Computation*, 113(2):173–198, 1994.
35. Taubenfeld G. and Moran S., Possibility and Impossibility Results in a Shared Memory Environment. *Acta Informatica*, 35:1–20, 1996.
36. Veríssimo P., Traveling Through Wormholes: Meeting the Grand Challenge of Distributed Systems. *Proc. Int. Workshop on Future Directions in Distributed Computing (FuDiCo)*, pp. 144–151, Bertinoro (Italy), June 2002.