A Simple Coding-Theory Based Characterization of Conditions for Solving Consensus

Roy Friedman
Department of Computer Science
Technion – Israel Institute of Technology
Haifa 32000
Israel
E-mail: roy@cs.technion.ac.il

June 30, 2002

Abstract

The condition based approach for solving consensus attempts to identify conditions on the input vectors for the distributed consensus problem that guarantee its safety and termination. This work presents a simple characterization for such conditions that permit consensus to be solved in an asynchronous network. Inspired by coding theory, this work views the input vectors as codes that encode the decision value. For the crash-failures model, this means that it is possible to solve consensus despite $f$ failures if and only if the allowed set of input vectors correspond to words of a code whose Hamming distance is at least $f + 1$. This is similar to recovering from erasure errors in coding theory. For the Byzantine model, consensus is solvable if and only if the allowed set of input vectors correspond to words of a code whose Hamming distance is at least $2 \cdot f + 1$. This is similar to overcoming bit flipping errors in coding theory. It is further shown that similar, less restrictive, codes can be used for solving the $k$-set consensus problem. The paper also discusses the practical implications of this characterization.

Keywords: Distributed Consensus, Coding Theory, Conditions for Consensus Solvability
1 Introduction

Distributed consensus is a fundamental problem in distributed computing, and much work has been done on investigating its solvability in various models, e.g. [10, 12, 13, 15, 17, 21, 30, 33]. In particular, it was shown that consensus cannot be solved in a purely asynchronous environment in which one process may fail by halting/crashing [21]. That is, for any algorithm that attempts to reach consensus in such an environment, there is a possible infinite run in which no process decides. This impossibility result is very disappointing, since only a single process is allowed to fail and in the most benign way. Also, the asynchronous distributed computing model is very attractive, since it does not impose any timing assumptions on the speed of processes or the delay of the network. Thus, problems that are solvable in asynchronous environments are solvable in other models of distributed computing as well. Similarly, asynchronous protocols can be developed without relying on timing assumptions, and are therefore more robust.

There are various techniques for circumventing the impossibility result, e.g., by presenting pseudo-synchronous models [18, 20], by introducing failure detectors [15], using randomized methods [7, 8], approximate agreement [19], and recently by constraining the allowed set of input vectors to the problem [33, 34, 35]. Specifically, in the latter approach, the initial values held by each process is viewed as an input vector to the consensus problem. It is then shown that in asynchronous environments, consensus is solvable despite $f$ failures when the set of allowed input vectors obey a certain family of conditions, which are shown to be necessary and sufficient.

This paper extends the work of Mostefaoui, Rajsbaum, and Raynal [33], by presenting a much simpler characterization of the conditions on input vectors to the consensus problem when only crash failures are allowed based on coding theory [9, 29]. Similar conditions on input vectors that permit consensus to be solved in asynchronous networks despite $f$ Byzantine failures are also presented. Specifically, we show that consensus can be solved if and only if the set of allowed input vectors correspond to a code in which the Hamming distance between any two code words leading to different decoded words is at least $f + 1$. (The Hamming distance of two vectors is defined as the number of entries by which the two vectors differ [9, 29].) Similarly, to overcome Byzantine failures, the input vectors must correspond to a code in which the Hamming distance of any two code words that represent two different decoded words is at least $2 \cdot f + 1$. We call the exact codes $f + 1$-admissible and $(2 \cdot f + 1)$-admissible, respectively.

Note that if the conditions hold, it is trivial to solve consensus. That is, if the distance between any two code words leading to two different decoded words is at least $f + 1$, then if each process sends its input bit to all other processes, each process will have at least $n - f$ valid input bits. By coding theory, each process can then decide on the correct decoded word. In the case of Byzantine failures, if we assume that there are at most $f$ failures, each process can wait until it has heard enough values such that the bits it has seen can be corrected to a legal code word by the error correcting mechanism corresponding to the code. (See Section 3 for more accurate details.)
The more difficult part is showing that consensus cannot be solved unless the allowed set of input vectors corresponds to such a code. The main difficulty here lies in the fact that in coding theory, each code word represents only a single decoded word, while in consensus there is no explicit requirement that every initial configuration is also a *univalent configuration*, i.e., a configuration that can only lead to a single value. In the paper we present two proofs for the claim. One is based on the proof of the consensus impossibility result that was shown in [2, 6]. In particular, as part of the proof in [2, 6], it is established that if there is a *bi-valent* initial configuration, then consensus cannot be solved. In our case, all we have to show is that unless the input vectors obey our criteria, there has to be a bi-valent initial configuration, and thus consensus cannot be solved. The other proof is a very minor adaptation of the proof in [33, 31], which has a combinatorial flavor.

The $k$-set consensus problem is a generalized form of the consensus problem [16]. Clearly, any code that can be used to solve consensus, can also be used to solve $k$-set consensus. Moreover, in this paper we define $(d,k)$-admissible codes, and show that $k$-set consensus is solvable when the input vectors belong to some $(f+1,k)$-admissible code in the crash-failure model, or to a $(2\cdot f+1, k)$-admissible code in the Byzantine model. This is shown by using roughly the same protocol presented here for the distributed consensus problem. We leave open the problem of showing the necessity of using $(f+1,k)$-admissible $(2\cdot f+1, k)$-admissible codes for solving the $k$-set consensus problem.

We also show in the Appendix that the characterization in [33] is equivalent to the strict version of ours, and that the specific conditions given there are special cases of our characterizations for crash-failures. Note that for consensus, we could have shown a direct equivalence between $(f+1)$-admissible codes and the notion of $f$-acceptability in [33]. However, we prefer to work directly with $d$-admissibility since the proofs are simpler, shed important insight about the relation between coding theory and consensus, and highlight the benefits of the coding theory approach. Also, the work of [33] does not address Byzantine failures. Finally, we discuss some practical implications of our results, and show that they also hold in the shared memory model.

In an independent, parallel, effort to this work, Mostefaoui, Rajsbaum, and Raynal have shown that the Interactive Consistency problem is equivalent to error correction [32]. Interactive Consistency is at least as strong as consensus, and as far as condition based solvability, it is in fact stronger. Specifically, the conditions identified in [32] for Interactive Consistency are similar to the ones we discuss here, except that they allow no redundancy, and are therefore more restrictive.

The rest of this paper is organized as follows: Section 2 presents the formal model and assumptions. Section 3 provides the characterization of the input vectors for solving consensus, and proves their sufficiency and necessity. Section 4 discusses the implications of guaranteeing some of the safety requirements even when the input vectors are not code words. Section 5 deals with $k$-set consensus, while Section 6 shows that the main results hold in the shared memory model as well. Section 7 discusses some practical implications of the work. Additional conclusions and discussions appear in Section 8.
2 Model

In this paper we assume the standard asynchronous message passing model [6, 27]. That is, we assume a set $N$ of $n$ processes, $n > 0$, communicating by exchanging messages over a communication network. The network is reliable, meaning that it eventually delivers every message sent from a live process to a process that does not crash. However, the latency of the network is unbounded. Also, we assume that processes do not have access to any global clock. We consider two subsets of the above model, based on the type of failures allowed. In the first model, called crash-failures, processes may fail by halting only. In the second model, called Byzantine-failures, process may fail arbitrarily, including by sending the wrong information, not sending a message at all, or deviating from their protocol in any possible way. However, we assume authentication, and thus, a process cannot impersonate as another process. A process that does not suffer a failure is called correct; otherwise, it is called faulty.

More formally, we can view each process as a deterministic automata, that starts in some initial state, and reacts to events it occurs by changing its local state and possibly sending messages by generating message-send events. The events that can occur in a process are startup, message-receive, and crash. We assume that each process that gets a message-receive event must get a startup event beforehand. A send event is written as message-send$(p_i, m)$ when the identity of the intended receiver $p_i$ and the contents of the message $m$ are important. Similarly, we write message-receive$(m)$ when the contents of the message received in a receive event is important. If a process incurs a crash event, this is the last event it experiences, and it is not allowed to send any additional messages. Naturally, message-receive events correspond to the delivery of a message from the network. In line with these definitions, we can define a history to be an initial state of a process and all following events it occurs and generates in the order they happen, and an execution to be a set of histories, one for each process. A history can be infinite, and an execution is infinite if some of its histories are infinite.

The reliability of the network is modeled by requiring that in every execution, each message-receive event has to correspond to a single message-send event. Also, if an execution $\sigma$ is infinite, then for each message-send$(p_j, m)$ event in $\sigma$, either $p_j$ incurs a crash event, or $p_j$ gets a message-receive$(m)$ event. Similarly, if process $p_k$ generated two sending events message-send$(p_j, m_1)$ and message-send$(p_j, m_2)$ in that order, then if $p_j$ incurs message-receive$(m_2)$ in $\sigma$ then it must also incur message-receive$(m_1)$ in $\sigma$. A protocol is a collection of transition functions, one for each process’ automaton.

In any given execution $\sigma$, we can define the causal history of a process $p$ at some prefix $HP_p$ of its history $H_p$ in $\sigma$ in the usual way, and denote it by $CH_p(\sigma, HP_p)$. Clearly, due to the determinism of each process, if there are two executions $\sigma_1$ and $\sigma_2$ and a history prefix $HP_p$ for some process $p$ such that $CH_p(\sigma_1, HP_p) = CH_p(\sigma_2, HP_p)$, then the state of $p$ at the end of $HP_p$ is the same in both $\sigma_1$ and $\sigma_2$. For a given execution, the initial configuration of the system is the set of initial states of all processes in the system.

We say that a process failed in the crash-failures model if its history includes a crash
event. Similarly, we say that a process failed in the Byzantine-failures model either if its history includes a crash event, or if its history includes state transitions and/or generation of events other than those specified by the protocol.

2.1 The Distributed Consensus Problem

The distributed consensus problem, hereafter, consensus, is defined as follows: We assume that each process $p_i$ holds an initial input bit whose value is $v_i$ taken from some domain $V$, and a decision value $u_i$ taken from the domain $V \cup \{\perp\}$, where $u_i$ is initially $\perp$. We say that a protocol solves the consensus problem if it guarantees the following properties:

**Validity:** A decision value $u_i \neq \perp$ has to be one of the initial input values $v_j$.

**Agreement:** All decisions values $u_i \neq \perp$ must be the same.

**Termination:** For every process $p_i$ that does not crash, eventually $u_i \neq \perp$.

**Irreversibility:** Each process can only change $u_i$ once.

As we mentioned before, it was shown in [21] that consensus cannot be solved in an asynchronous system even if one process suffers a crash failure. Similarly to [33], in this work we view the initial input bits of all processes as a vector $v^n$, and are interested in restricting the set of allowed input vectors. Let $V^n$ be the set of all possible input vectors, and let $C \subseteq V^n$ be a set of 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:

**C_Viability:** If the input values obey a condition $C$, then a decision value $u_i \neq \perp$ has to be one of the initial input values $v_j$.

**C_Agreement:** If the input values obey a condition $C$, then all decisions values $u_i \neq \perp$ of correct process must be the same.

**C_Guaranteed Termination:** If at most $f$ processes fail and the input values obey a condition $C$, then for every correct process $p_i$, eventually $u_i \neq \perp$.

**C_Irreversibility:** Each correct process can only change $u_i$ once.

For Byzantine failures we use the following definition of validity:

**CB_Viability:** If all input values are $v$, then every decision value $u_i \neq \perp$ of a correct process is equal to $v$. 

4
3 Characterizing Input Vectors for Solving Consensus

We define the initial configuration $c$ of the system as bi-valent if there are two executions $\sigma_1$ and $\sigma_2$ that start with $c$ such that the decision value in $\sigma_1$ is $v_1$, the decision value in $\sigma_2$ 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 code words coding the possible decision values. Clearly, in the case of consensus, since the decision value has to be unique, the code maps words from $V^n$ to words in $V$. Also, due to the validity requirement of consensus, we must limit ourselves to codes in which at least one of the bits of every code word corresponding to a word $v \in V$ has to be $v$. Thus we have:

**Definition 3.1:** A $d$-admissible code is a mapping $C : V^n \rightarrow V$ such that the Hamming distance of every two code words coding different words in $V$ is at least $d$ and at least one of the bits in each code word mapped to a word $v \in V$ is $v$.

### 3.1 Solving Consensus with $d$-Admissible Codes

We now present a simple generic protocol for solving the consensus problem using $d$-admissible codes in both the crash-failure and Byzantine-failure models in Figure 1.\(^1\) Of course, for the crash-failures model with at most $f$ failures, it is sufficient to use an $(f+1)$-admissible code, while for the Byzantine-failures model with at most $f$ failures we have to use a $(2f+1)$-admissible code. Also, the protocol uses a subroutine called `match` to check whether the bits received so far can be matched to any code word. It returns $\bot$ if no matching was found; otherwise, it returns the value of the decoded word corresponding to the code word matched.

More precisely, we treat each bit that was not received from some process as the special value $\bot$ in the vector of received bits. Then, all `match` has to do is to check if the distance of this vector is at most $f$ from some legal code word. If it is, `match` returns the value mapped to by this code word. Otherwise, `match` returns $\bot$. 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 code word. Also, it is possible that there would be several possible code words to choose from, but in this case they all have to be mapped to the same value, so any of them can be picked. 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. Mostefaoui, Rajsbaum, Raynal, and Roy explored the relation between the specification of a condition and the complexity of the equivalence of the `match` routine in their work, and described a hierarchy of conditions based on this tradeoff [34].

\(^1\)This protocol is similar to the first four lines of the condition-based consensus protocol in [33].
Variables:
  \( v_i \) - initial input value
  \( u_i \) - decision value
  \( V_i \) - Vector of initial values heard so far
  \( \text{code} \) - the code used by the allowed input values

Initially:
  \( u_i := \perp; \)
  \( v_i := \text{some value from } V; \)
  \( \forall j \in [1\ldots n] \text{ do } V_i[j] := \perp \)

Start:
  Generate \( \text{send-message}(p_j, v_i) \) to every process \( p_j \) including myself

Upon \( \text{receive-message}(v_j) \) sent by \( p_j \):
  \( V_i[j] := v_j; \)
  \( \text{val} := \text{match}(V_i; \text{code}); \)
  if \( \text{val} \neq \perp \) then
    \( u_i := \text{val} \)
  endif

Figure 1: Generic protocol for solving consensus using \( d \)-admissible codes – code for process \( p_i \)

Also, the protocol in Figure 1 provides validity, agreement, and termination when the input vectors are indeed code words of some \((f + 1)\)-admissible code. We discuss in Section 4, the implications of satisfying validity and agreement when the input vectors are not always code words, and termination when the input vectors are not code words but there are no failures.

### 3.2 Necessity of \( d \)-Admissibility for Solving Consensus

**Theorem 3.1:** Let \( C \) be a condition on the allowed set of input vectors for the consensus problem, and let \( P \) be a protocol. If \( P \) is an \( f \)-fault tolerant protocol for solving consensus for condition \( C \) in the crash-failures model, then \( C \) consists of the code words of an \((f + 1)\)-admissible code.

**Proof:** To prove the theorem, we first point the reader to the proof of the consensus impossibility result in [2, 6]. 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 \( f \) failures is completely analogous. Thus, all that we have to show is that if the initial input vectors allowed by the condition are not words of an \((f + 1)\)-admissible code, then there has to be a bi-valent initial configuration.

Assume by way of contradiction that \( C \) does not correspond to an \((f + 1)\)-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 $f + 1$ processes, yet each one leads to a different value $v_1$ and $v_2$ respectively. Denote by $P' = p_{i_1}, \ldots, p_{i_k}$ ($k \leq f$) the processes that differ between $c_1$ and $c_2$. Thus, there is an execution $\sigma_1$ of the protocol that starts at $c_1$ in which all processes in $P'$ fail before managing to take any action, and thus all other processes decide on value $v_1$ without receiving any message from any process in $P$. Let $p_j$ be one of the processes that decide in $\sigma_1$ 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 $\sigma_2$ that starts in $c_2$, no process fails during $\sigma_2$, but $CH_{p_j}(\sigma_1, HP_{p_j})$ is also a causal history of $p_j$ in $\sigma_2$. Given the determinism of $p_j$, it must also decide in $\sigma_2$ 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.

**Theorem 3.2:** Let $C$ be a condition on the allowed set of input vectors for the consensus problem, and let $\mathcal{P}$ be a protocol. If $\mathcal{P}$ is an $f$ fault tolerant protocol for solving consensus for condition $C$ in the Byzantine-failures model then $C$ consists of the code words of an $(2 \cdot f + 1)$-admissible code.

**Proof:** To prove the theorem, we first note that the proof in [2, 6] 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 $(2 \cdot f + 1)$-admissible code, then there has to be a bi-valent initial configuration.

Assume by way of contradiction that $C$ does not correspond to a $(2 \cdot f + 1)$-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 $2 \cdot f + 1$ 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 $\sigma_1$ of the protocol that starts at $c_1$ in which all processes in $P'$ crash before managing to take any action, and thus all other processes decide 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 decide in $\sigma_1$ 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 that following execution $\sigma_2$ that starts in $c_2$. In $\sigma_2$, 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 $\sigma_2$. 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 $\sigma_2$. Given the determinism of $p_j$, it must also decide in $\sigma_2$ 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.
4 Strict Condition Based Consensus

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

4.1 Agreement

Variables:
- $v_i$ - initial input value
- $w_i$ - intermediate decision value
- $u_i$ - decision value
- $V_i$ - Vector of initial values heard so far
- $W_i$ - Vector of intermediate decision values heard so far
- code - the code used by the allowed input values

Initially:
- $u_i := \bot$; $v_i :=$ some value from $V$
- $\forall j \in [1 ... n]$ do $V_i[j] := \bot$

Start:
- Generate send-message($p_j, v_i$) to every process $p_j$ including myself

Upon receive-message($v_j$) sent by $p_j$:
- $V_j[j] := v_j$
- $val := \text{match}(V_i, \text{code})$
- if $val \neq \bot$ then
  - $w_i := val$
  - Generate send-message($p_j, w_i$) to every process $p_j$ including myself
- endif

Upon receive-message($w_j$) sent by $p_j$:
- $W_i[j] := w_j$
- if the value of $w_j$ appears at least $n - f$ times in $W_i$ then
  - $u_i := w_j$
- endif

Figure 2: A protocol that always ensures Agreement when $f < \frac{n}{2}$ in the crash-failure model and when $f < \frac{n}{3}$ in the Byzantine model - code for process $p_i$.

In order to ensure that agreement holds even when the input vectors are not legal code words, we augment our original protocol with additional checks that verify that all decision values are
the same. The modified protocol is depicted in Figure 2; it guarantees Agreement whenever $f < \frac{n}{3}$ in the benign failures model and when $f < \frac{n}{2}$ in the Byzantine model. The protocol also guarantees C.Validity and C.Guaranteed Termination.

4.2 Validity

Even the protocol of Figure 2 might not satisfy Validity when the input vector is not a legal code word. To overcome this problem, we define a slightly stronger definition of admissibility, namely:

**Definition 4.1:** A *strongly $d$-admissible code* is a mapping $C : \mathcal{V}^n \rightarrow V$ such that the Hamming distance of every two code words coding different words in $\mathcal{V}$ is at least $d$ and at least $d$ of the bits in each code word mapped to a word $v \in V$ are $v$.

We say that an $f$-fault tolerant protocol solves the strict consensus problem for a given condition $C$ if it guarantees the following properties: Validity, Agreement, C.Guaranteed Termination, and C.Irreversibility. Clearly, the protocol of Figure 2 solves strict consensus for strongly $(f+1)$-admissible codes. Also, with binary alphabet, any $d$-admissible code that includes the word $0, \ldots, 0$ is also strongly $d$-admissible, and in these cases the problems are identical.

**Theorem 4.1:** Let $C$ be a condition on the allowed set of input vectors for the consensus problem, and let $P$ be a protocol. If $P$ is an $f$-fault tolerant protocol for solving strict consensus for condition $C$ in the crash-failures model, then $C$ consists of the code words of a strongly $(f+1)$-admissible code.

**Proof:** First, note that the proof of Theorem 3.1 holds here as well. The only thing we need to show is that Validity cannot be guaranteed unless the code satisfies the property that each word is mapped to a value that appears in at least $f + 1$ of its bits. Assume by way of contradiction that there is a protocol $P$ that solves strict consensus for a condition $C$ that includes an allowed input vector in which no value appears more than $k \leq f$ times. In other words, every execution of $P$ that starts with $V$ has to terminate.

As discussed before, since $P$ solves consensus for $C$, $V$ must be an initial univalent configuration. Consider an execution of $P$ that starts with $V$ and decides some value $v$. Thus, every execution of $P$ that starts with $V$ must decide $v$. Since $v$ only appears $k \leq f$ 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 $S$. Thus, 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 therefore $E'$ also terminates with a decision value $v$. However, this violates Validity.

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.
4.2.1 A Combinatorial Proof of Theorem 4.1

The combinatorial proof of Theorem 4.1 follows closely the proof of a similar theorem in [33]. That is, we start by defining the notion of \( f \)-legality. We then show that if the consensus problem for a condition \( C \) is \( f \)-fault tolerant solvable, then \( C \) is \( f \)-legal. Finally, we show that any condition \( C \) that is \( f \)-legal, is also \((f + 1)\)-admissible.

To present the proof, we need to add some notation. Given a vector \( I \), we define \( I_f \) to be the set of vectors obtained by replacing up to \( f \) entries in \( I \) with \( \perp \), and denote by \( C_f \) the union of all \( I_f \) for every \( I \in C \). Given two vectors \( J_1 \) and \( J_2 \), \( J_1 \leq J_2 \) iff for each \( j \) such that \( J_1[j] \neq \perp \), \( J_1[j] = J_2[j] \).

Given the above definitions, and a condition \( C \), we construct a graph \( Gin(C, f) \) as follows: The set of vertices is \( C_f \). Also, every two vertices \( J_1 \) and \( J_2 \) are connected in \( Gin(C, f) \) iff \( J_1 \leq J_2 \). Note that this implies that every two vectors whose Hamming distance is at most \( f \) are connected by a path in \( Gin(C, f) \). Moreover, \( Gin(C, f) \) is made up of one or more connected components.

**Definition 4.2:** A condition \( C \) is \( f \)-legal if for each component of \( Gin(C, f) \) all vertices belonging to this component have at least one input value in common.

Then, the following lemma, which is the same as Lemma 5.4 in [33], can be proved verbatim to its proof in [31].

**Lemma 4.2:** If a consensus problem for a condition \( C \) is \( f \)-fault tolerant solvable, then \( C \) is \( f \)-legal.

The only thing left to be shown is the following lemma:

**Lemma 4.3:** Any condition \( C \) that is \( f \)-legal consists of code words of an \((f + 1)\)-admissible code.

**Proof:** We need to show that the vectors in \( C \) are words of a code in which the Hamming distance of every two words leading to different decoded words is at least \( f + 1 \). Clearly, all vectors in \( C \) whose Hamming distance is less than \( f + 1 \) are in the same connected component, and the Hamming distance of every two vectors that are in separate connected components must be at least \( f + 1 \). By the assumption that \( C \) is \( f \)-legal, at least one value appears in all nodes of each such connected component. We can then choose the code to be the mapping of all vectors with no \( \perp \) elements to one of the common values in their connected component. Next, we claim that if two code words map to different values, then their Hamming distance is at least \( f + 1 \). To see this, note that by the way the code is chosen, if two code words map to different values, then they are in different connected components, and therefore their Hamming distance is at least \( f + 1 \). \( \square \)
4.3 Termination

To guarantee termination in the crash-failure model with $f < \frac{n}{2}$, both when the input vector is a code word and when there are no failures (but the input vector is not a code word), it is possible to use the protocols in [31]. At present, we have no solution for the Byzantine model.

5 $k$-Set Consensus

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

**CK\_Validity:** If the input values obey a condition $C$, then a decision value $u_i \neq \perp$ has to be one of the initial input values $v_j$.

**CK\_Agreement:** If the input values obey a condition $C$, then there can be no more than $k$ distinct decision values $u_i \neq \perp$.

**CK\_Guaranteed Termination:** If at most $f$ processes fail and the input values obey a condition $C$, then for every process $p_i$ that does not fail, eventually $u_i \neq \perp$.

**CK\_Irreversibility:** Each correct process can only change $u_i$ once.

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

**CKB\_Validity:** If the initial value of all processes is the same, then every correct process that decides has to decide on this value.

**CKB\_Agreement:** If the input values obey a condition $C$, then there can be no more than $k$ distinct decision values $u_i \neq \perp$ among the correct processes.

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

**Definition 5.1:** A $(d, k)$-admissible code is a mapping $C : V^n \rightarrow V$ such that: (a) for every code word $w$ in $C$, all code words whose Hamming distance from $w$ is less than $d$ are mapped to at most $k$ different values, and (b) at least one of the bits in each code word mapped to a word $v \in V$ is $v$.

Given the above definition, we claim that $k$-set consensus can be solved if the input vectors to the problem are code words of a $(f + 1, k)$-admissible code in the crash-failures model, and $(2 \cdot f + 1, k)$-admissible code in the Byzantine model. Note that by slightly changing the behavior of the match routine in the protocol in Figure 1, 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 bits from any code word is at most $f$. If the answer is yes, match returns the value pointed to by the closest of these code words, breaking symmetry arbitrarily. Otherwise, match returns $\perp$. 

1
6 Shared Memory

6.1 Shared Memory Model

The model here is the same as in Section 2, with the following exceptions: A shared memory consists of a collection of objects $O$. Also, instead of send and receive events, a history of a process includes start-read($o$), end-read($v$), start-write($o$, $v$), and end-write events. In these events, $o$ is some object from $O$, and $v$ is some value from a given range $V$. We assume that all process histories are sequential, or in other words, the event that immediately follows a start-read($o$) in the history can either be end-read($v$) or crash. Similarly, the event that immediately follows a start-write($o$, $v$) in the history can either be end-write or crash. The combination of start-read($o$) and the immediately following end-read($v$) forms a read($o$) operation, which reads the value $v$ from $o$. The combination of start-write($o$, $v$) and the immediately following end-write or crash event forms a write($o$, $v$) operation, which writes $v$ to $o$. Note that this also defines an obvious order on the operations of a history.

For a process history $H_p$, we define a sequence of operations $S_p$ to be a serialization of $H_p$ if $S_p$ includes the same operations as $H_p$, and the operations are ordered in $S_p$ in the same order as in $H_p$. For a given sequence of operations $S$, we define $S|_p$ to be the projection of $S$ to operations of process $p$ only. For a given execution $\sigma$, we define a serialization of $\sigma$ to be a sequence of operations $S$ such that for each process $p$, $S|_p = S_p$, where $S_p$ is the serialization of $p$’s history. Also, we define a legal serialization to be a serialization in which every read operation $r$ reads the last value written to the object it is reading from. In this work, we assume that for each execution, there is a legal serialization of $\sigma$, or in other words, that each execution obeys the sequential consistency condition [25]. (In fact, it is easy to verify that the results also hold for some weaker memory models, such as PRAM [26] and causal memory [1].)

6.2 Sufficiency of $d$-Admissibility in Shared Memory

Figure 3 gives a modified version of the protocol in Figure 1 that solves consensus in the shared memory model whenever the allowed input vectors correspond to an $(f + 1)$-admissible code. Moreover, by assuming that a Byzantine process cannot write to a variable of another process, or prevent a non-faulty process from reading a value of another non-faulty process, the protocol of Figure 3 solves Byzantine consensus whenever the allowed input vectors correspond to a $(2 \cdot f + 1)$-admissible code. This is true even if Byzantine processes can erase a value they have already written, or somehow write their own entries such that different processes read different values from them. The above can happen either if a Byzantine process writes its entry multiple times and each time writes a different value, or if it interferes with the low level mechanisms that implement the shared memory.
Variables:
- $v_i$ – initial input value
- $u_i$ – decision value
- $V$ – shared vector of values, each entry can be written by the corresponding process and read by all processes.
  All entries are initialized to $\perp$
- $V_i$ – local version of $V$
- code – the code used by the allowed input values

Initially:
- $u_i := \perp$; $v_i :=$ some value from $V$;
- $\forall j \in [1 \ldots n]$ do $V_i[j] := \perp$

Start:
- $\text{write}(V[i], v_i)$

Periodically:
- $\forall j \in [1 \ldots n]$ do $V_i[j] := \text{read}(V[j])$
- $val := \text{match}(V_i[,\text{code}])$
- if $val \neq \perp$ then
  - $u_i := val$
  endif

Figure 3: Generic protocol for solving consensus in shared memory model using $d$-admissible codes – code for process $p_k$
6.3 Necessity of \(d\)-Admissibility in Shared Memory

Attiya et al. showed a simulation of shared memory on top of message passing that is resilient to \(f < \frac{n}{2}\) benign failures [4]. Thus, it is trivial to show that consensus cannot be solved if the allowed set of initial input vectors does not correspond to an \((f + 1)\)-admissible code. Otherwise, we could have used the simulation of [4] and contradict Theorem 3.1. Similarly, Malkhi and Reiter have shown a construction of read-write registers that can survive \(f < \frac{n}{3}\) Byzantine failures, e.g., when the quorums are any collection of \(\frac{2n}{3}\) processes [28]. Thus, the necessity result holds in the shared memory model for the Byzantine case as well.

7 Practical Implications of the Results

The practical implication of our work 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 to 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 of Figure 1. If the size of the smallest cluster is at least \(f + 1\) \((2 \cdot f + 1)\), we can solve consensus in the global system despite \(f\) failures.

A shortcoming of this approach is that if a cluster becomes disconnected, the protocol of Figure 1 will block until the cluster reconnects 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 bit-flip errors, there are more efficient codes. For example, parity can be used to overcome one bit erasure with only one extra bit, while Hamming code can correct one bit flip with an overhead of \(\log(n)\) bits. But, in both cases, some bits in each code word depend on many other bits in the same word. Practically speaking, given a code, if some bit \(b\) depends on the values of some other bits \(b_1, \ldots, b_l\), 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 bits each bit depends on, and the bit overhead for error correction. That is, small overhead implies the ability to solve consensus with small hardware redundancy, while low dependency between bits 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 [9, 29].

Note that Plützmann 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 [37]. Also, Fitzi and Maurer showed how to obtain Global Broadcast in the presence of up to \(\frac{n}{3}\) Byzantine failures based on a Local Broadcast service [22]. However, none of these works draws any relation from agreement to error correction.
7.1 Cluster Based Failure Detectors

The discussion above also implies that it is possible to solve consensus despite a *small number of failures* using failure detectors that provide the accuracy and completeness properties of ◇W (or ◇S) 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 [15] as follows:

**Strong ◇-Completeness:** Eventually, every process that fails is permanently suspected by every non-faulty process in the same cluster.

**Weak ◇-Completeness:** Eventually, every failed process is permanently suspected by some non-faulty process in the same cluster.

**Eventual Strong ◇-Accuracy:** There is a time after which no non-faulty process is suspected by any non-faulty process in the same cluster.

**Eventual Weak ◇-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 [15], 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 ◇◇W if it guarantees Weak ◇-Completeness and Eventual Weak ◇-Accuracy. Similarly, a failure detector belongs to the class ◇◇S if it guarantees Strong ◇-Completeness and Eventual Weak ◇-Accuracy.

Clearly, it is possible to simulate a failure detector in ◇◇S from a failure detector in ◇◇W by running within each cluster the same simulation that was given in [15] for simulating ◇S from ◇W. Thus, it is possible to solve consensus among members of the same cluster using ◇◇S (and therefore ◇◇W) and any of the consensus protocols that is based on ◇S or ◇W, e.g., [15, 24, 36, 38]. 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 1. We call this the *direct cluster based approach*.

On the other hand, it is easy to derive a failure detector in ◇W from a failure detector in ◇◇W. Specifically, assume that each process is equipped with a failure detector FD from the class ◇◇W.

**Claim 7.1:** 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 ◇W.

**Proof:** Note that FD' ∈ ◇◇W, since it behaves the same as FD for processes inside the same cluster. Clearly, Weak ◇-Completeness is stronger than Weak Completeness. This is
because the latter only requires that eventually, every failed process is 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 $\diamond W$.

Therefore, we can employ Claim 7.1 to simulate a failure detector in $\diamond W$, and use it to solve consensus with any of the protocols of [15, 24, 36, 38]. 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. Similarly, the simulation of $\diamond S$ from $\diamond W$ that was given in [15] requires many long haul message exchanges. Moreover, with the direct cluster based approach, all rounds of $\diamond S$ 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, e.g., group membership [23], 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. In contrast, existing protocols for solving consensus (w.r.t. the entire set of nodes) that rely on $\diamond S$ can overcome up to $\left\lfloor \frac{n}{3} \right\rfloor - 1$ failures.

8 Discussion

In this paper we have identified a very simple characterization for conditions on input vectors for solving the consensus problem, inspired by results from coding theory. It is based on the observation that consensus cannot be solved if there are possible bi-valent initial configurations, and thus the decided value has to be determined uniquely based on the initial configuration. This means that in order to solve consensus, the allowed initial configurations for the consensus problem can be treated as codes, and thus crash-failures can be compared to erasure errors and Byzantine-failures to bit-flip errors in coding theory. We have introduced the notion of $d$-admissible codes, based on the minimal Hamming distance of code words that lead to different decoded words, and showed that consensus is solvable in the crash-failures (Byzantine-failures) model if and only if the allowed initial input values correspond to an $(f+1)$-admissible $((2 \cdot f+1)$-admissible) code.

In a similar manner, we defined $(d, k)$-admissible codes, and showed that $k$-set consensus can be solved if the input vectors are guaranteed to be words of a $(f+1, k)$-admissible $((2 \cdot f+1, k)$-admissible) code. As mentioned before, we left the question of whether any condition that
allows solving $k$-set agreement when $k < f$ is at least $(f + 1, k)$-admissible ($(2 \cdot f + 1, k)$-admissible). Following our work, Attiya and Avidor have given a necessary and sufficient topological characterization of the input vectors to the wait-free case of the set consensus problem [3]. Similarly, looking at other related agreement problems, such as primary partition group membership [11, 14], atomic commit [39], and naming [5] in the context of coding theory would also be interesting.

We have presented simple protocols that solves consensus whenever the initial input values indeed form a $(f+1)$-admissible $(2 \cdot f + 1)$-admissible code, in which each process sends one message to all other processes. These protocols do not guarantee termination if the input vector is not a code word, even when there are no failures. To guarantee termination in these conditions for the crash failure model with $f < \frac{n}{2}$, it is possible to use the protocol in [31]. Finding a protocol that always guarantees agreement, and also guarantees termination both when the input vector is in a $(2 \cdot f + 1)$-admissible code and when there are no failures (but the input vector is not a code word) for the Byzantine case is still an open problem. We have also discussed the implications of trying to enforce agreement and validity even when the input vectors are not code words. For validity, this means stricter codes, and for agreement, a more expensive protocol.

We have presented a direct proof for the necessity of $d$-admissible codes for both the crash-failures model and the Byzantine-failures model, and a combinatorial proof only for the strict version of the problem in the crash-failures model. It would be interesting to extend the combinatorial proof to the Byzantine-failures model as well. Similarly, in recent years, several topological proofs for agreement problems, including consensus, have been introduced, e.g., [12, 13]. Finding a topological proof for $d$-admissible codes would be interesting.

In general, we believe that finding a linkage between coding theory and agreement problems in distributed computing is an important goal. Coding theory is an area that was studied extensively. 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, one can already draw the immediate conclusion that it is impossible to compute error correcting data, e.g., parity bit, in asynchronous distributed environments in which a single node can fail.

Acknowledgements: I would like to thank Hagit Attiya for many insightful discussions and to Sergio Rajsbaum for finding an error in a previous version of this paper. Thanks also to Michel Raynal and Ronny Roth for helpful comments.
References


A  \( f \)-Acceptability vs. Strong \((f + 1)\)-Admissibility

The notion of \( f \)-acceptability was defined in [33]. Given our results using strongly \((f + 1)\)-admissible codes, and the results in [33] that an \( f \)-fault tolerant protocol solves consensus iff the condition on input vectors is \( f \)-acceptable, it is clear that the two notions are equivalent. To further the reader’s insight about this, we show here directly that the two definitions are indeed equivalent, and also discuss conditions \( C_1 \) and \( C_2 \) of [33] in the context of codes.

The definition of \( f \)-acceptability requires the existence of a predicate \( P \) and a function \( S \) with the following properties:

- Property \( T_{C \rightarrow P} \): \( I \in C \Rightarrow \forall J \in I_f : P(J) \).
- Property \( A_{P \rightarrow S} \): \( \forall J_1, J_2 \in V^f_j : (J_1 \leq J_2) \land P(J_1) \land P(J_2) \Rightarrow S(J_1) = S(J_2) \).
- Property \( V_{P \rightarrow S} \): \( \forall J \in V^f_j : P(J) \Rightarrow S(J) = \) a non-\( \perp \) value of \( J \).

**Definition A.1:** A condition \( C \) is \( f \)-acceptable if there exists a predicate \( P \) and a function \( S \) satisfying properties \( T_{C \rightarrow P} \), \( A_{P \rightarrow S} \), and \( V_{P \rightarrow S} \) for \( f \).

When looking at the above definition, \( P \) can be thought of as a predicate that evaluates to \text{true} whenever a word can be safely extended to a code word, and \( S \) is the decoded word corresponding to the corrected code word. It is easy to see that any strongly \((f + 1)\)-admissible code obeys the definition of \( f \)-acceptability, and all vectors in an \( f \)-acceptable condition are also code words of some strongly \((f + 1)\)-admissible code. In particular, the Hamming distance of any two vectors \( I_1 \) and \( I_2 \) in an \( f \)-acceptable condition for which \( S(I_1) \neq S(I_2) \) must be at least \( f + 1 \) in order to preserve Property \( A_{P \rightarrow S} \).

Condition \( C_1 \) in [33] 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 [33] 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 \( f + 1 \).