Title
Consensus for Dependent Process Failures

Authors
Junqueira, Flavio
Marzullo, Keith

Publication Date
2003-02-18

Consensus for Dependent Process Failures [*]

Flavio P. Junqueira
`flavio@cs.ucsd.edu`

Keith Marzullo
`marzullo@cs.ucsd.edu`

University of California, San Diego
Department of Computer Science and Engineering
9500 Gilman Drive
La Jolla, CA

8th October 2002

Keywords: Distributed Systems, Fault Tolerance, Correlated Failures, Consensus

---

[*] This work was developed in the context of the RAMP project, supported by DARPA as project number N66001-01-1-8933.

1 Introduction

Most fault-tolerant protocols are designed assuming that out of n components, no more than t can be faulty. For example, solutions to the Consensus problem are usually developed assuming no more than t of the n processes are faulty where "being faulty" is specialized by a failure model. We call this the *t of n assumption.* It is a convenient assumption to make. For example, bounds are easily expressed as a function of t: if processes can fail only by crashing, then the Consensus problem is solvable when $t < n$ if the system is synchronous and when $t < 2n$ if the system is asynchronous extended with a failure detector of the class $\diamond W$. [1, 2]

The use of the t of n assumption dates back to the earliest work on fault-tolerant computing. [3] It was first applied to distributed coordination protocols in the SIFT project [4] which designed a fly-by-wire system. The reliability of systems like this is a vital concern, and using the t of n assumption allows one to represent the probabilities of failure in a simple manner. For example, if each process has a probability p of being faulty, and processes fail independently, then the probability $P(t)$ of no more than t out of n processes being faulty is:

$$P(t) = \sum_{i=0}^{t} \binom{n}{i} p^i (1-p)^{n-i}$$

If one has a target reliability R then one can choose the smallest value of t that satisfies $P(t) \geq R$.

The t of n assumption is best suited for components that have identical probabilities of failure and that fail independently. For embedded systems built using rigorous software development this is often a reasonable assumption, but for most modern distributed systems it is not. Process failures can be correlated because, for example, the same buggy software was used. [5] Computers in the same room are subject to correlated crash failures in the case of a power outage.

That failures can have different probabilities and can be dependent is not a novel observation. The continued popularity of the t of n assumption, however, implies that it is an observation that is being overlooked by protocol designers. If one wishes to apply, for example, a Consensus protocol in some real distributed system, one can use one of two approaches:

1. Use some off-line analysis technique, such as fault tree analysis [6] to identify how processes fail in a correlated manner. For those that do not fail independently or fail with different probabilities, re-engineer the system so that failures are independent and identically distributed (IID).

2. Use the same off-line analysis technique to compute what the maximum number of faulty processes can be, given a target reliability. Use this value for t and compute the value of n that, under the t of n assumption, is required to implement Consensus. Replicate to that degree.

Both of these approaches are used in practice. [6] This paper advocates a third approach:

3. Use the same off-line analysis to identify how processes fail in a correlated manner. Represent this using our abstraction for dependent failures, and replicate in a way that satisfies our replication requirement and that minimizes the number of replicas. Instantiate the appropriate dependent failure protocol.

We believe that our approach and protocols are amenable to on-line adaptive replication techniques as well.

In this paper we propose an abstraction that exposes dependent failure information for one to take advantage of in the design of a protocol. Like the t of n assumption, it is expressed in a way that hides its underlying probabilistic nature in order to make it more generally applicable.

We then apply this abstraction to the Consensus in both synchronous and asynchronous models assuming crash and arbitrary failures. We show replication requirements that are sufficient to enable a solution for Consensus. In order to demonstrate sufficiency, we applyed simple modifications to Consensus algorithms proposed in the literature. Although we cannot generalize this result to every problem in fault-tolerant distributed computing, we believe that our work does not invalidate all the previous work assuming t of n process failures. We also show that expressing process failure correlations with our model enables the solution of Consensus in some systems in which it is impossible when making the t of n assumption.

There has been some work in providing abstractions more expressive than the t of n assumption. The hybrid failure model (for example, [7]) generalizes the t of n assumption by providing a separate t for different classes of failures. Using a hybrid failure model allows one to design more efficient protocols by having sufficient replication for masking each failure class. It is still based on failures in each class being independent and identically distributed. In this paper, however, we do not consider hybrid failure models.

Byzantine Quorum systems have been designed around the abstraction of a *Fail-prone System* [8]. This abstraction allows one to define quorums that take correlated failures into account. This abstraction has been used to express a sufficiency condition for replication. Our work can be seen as generalizing this work, which applies only to Quorum Systems.

The remainder of this paper is divided as follows. Section 2 presents our assumptions for the system model and introduces our abstraction that models dependent process failures. Section 3 defines the distributed Consensus problem. Sections 4 and 6 present replication requirements and algorithms for synchronous Consensus on the crash and arbitrary failure models, respectively. For asynchronous Consensus, replication requirements and algorithms on the crash and arbitrary failure models are presented in sections 5 and 7, respectively. Finally, we draw conclusions and discuss future work in Section 8.

2 System Model

A system is composed of a set Π of processes, numbered from 1 to $n = |\Pi|$. The number assigned to a process is its process *id*, and it is known by all the other processes. In the rest of paper, every time we refer to a process with id i, we use the notation p_i. Additionally, we define Pid as the set of process id's, i.e., $Pid = \{i : p_i \in \Pi\}$. We use this set to define a sequence w of process id's. Such a sequence w is an element of Pid^*.

A process communicate with others by exchanging messages. Messages are transmitted through point-to-point reliable channels, and each process is connected to every other process through one of these channels. We model a channel between processes p_i and p_j as two pairs of buffers: $input_{ij}/output_{ij}$ and $input_{ji}/output_{ji}$. If process p_i sends a message m to p_j, then it places$m$ at buffer $input_{ij}$. Once the transfer of the message is completed, according to the timing assumptions, the message is moved to $output_{ij}$. Process p_j then has access to m. Note that process p_i only has control over the buffers $input_{ij}$ and $output_{ji}$.

Processes, on the other hand, are not assumed to be reliable. We consider both crash and arbitrary process failures. Different from most previous works in fault-tolerant distributed systems, process failures are allowed to be correlated. We introduce a new abstraction, namely *core*, which corresponds to a reliable subset of processes. From a set of cores, it is possible to derive subsets of processes such that in every run of the system at least one of these subsets contains only correct processes. We call them survivor sets.

Each process $p \in \Pi$ executes a deterministic automaton as part of the distributed computation [2, 9]. A deterministic automaton is composed of a set of states, a initial state, and a transition function. The collection of the automata executed by the processes is defined as a distributed algorithm. An execution of a distributed algorithm proceeds in steps of the processes. In a step, a process may: 1) receive a message; 2) undergo a state transition; 3) send a message to a single process. Steps are assumed to be atomic, and there is no restriction in terms of sequentiality. That is, steps of different processes are allowed to overlap in time. A process is assumed to take a step at global time $t \in \mathcal{T}$ provided by some external device. Although processes do not have access to this external device, this assumption turns out to be useful in reasoning about the systems we discuss here. The range of $\mathcal{T}$ is the non-negative integers.

Although the computational model is the same independently of the timing assumptions, we describe algorithms for synchronous and asynchronous systems differently. As we show later in this section, we explore the fact that the computation can be split in synchronous rounds to facilitate the coordination among the processes.

This is the general picture of our system model. In the following subsections, we discuss in details its various aspects.

2.1 Processes, Cores, and Survivor Sets

A system is composed of a set $\Pi = \{p_1, p_2, \cdots, p_n\}$ of processes. In our model, process failures are allowed to be correlated, which means that the failure of a process may indicate an increase in the failure probability of another process. To represent these correlations, we assume some abstraction. For example, processes can be represented by attributes and processes sharing an attribute have higher probability of failing in the same execution of the system.

To achieve fault-tolerance in a system assuming no failed process recovers, it is necessary to guarantee that non-empty subsets of Π survive to every execution. A process is said to survive to an execution if and only if it is correct in that execution. Thus, we would like to distinguish subsets of processes such that the probability of all processes in each of these subsets failing is negligible. Moreover, we want these subsets to be minimal in that removing any process of such a subset c makes the probability of all the processes in c failing non-negligible, These subsets are called *cores*. Cores can be extracted from the information about process failure correlations. In this paper, however, we assume that the set of cores is provided as part of the system specification. Models to describe failure correlations and methods to extract cores from instances of these models are not addressed here.

By assumption, each core contains at least one process that is going to be correct in an execution. Thus, a subset of processes, such that the intersection with every core is non-empty contains processes that are correct in some execution. If such a subset is minimal, then it is called a survivor set. Notice that in every run of the system there is at least one survivor set that contains only correct processes. The definition of survivor sets is equivalent to the one of a *fail-prone system* $\mathcal{B}$ [8]. The set of all survivor sets is the complement of $\mathcal{B}$.

We now define cores and survivor sets more formally. Let R be a rational number expressing the target degree of reliability for Π, and $r(x)$, $x \subseteq \Pi$, be a function that evaluates to the reliability of the subset x. We define cores and survivor sets as follows:

Definition 2.1 Given a set of processes Π and target degree of reliability $R \in [0,1] \cap Q$, c is said to be a *core* if and only if:

1. $c \subseteq \Pi$;
2. $r(c) \geq R$;
3. $\forall p \in c$, $r(c - \{p\}) < R$.

C_Π is the set of cores of Π. Given a set of processes Π and a set of cores C_Π, s is said to be a survivor set if and only if:

1. $s \subseteq \Pi$;
2. $\forall c \in C$, $s \cap c \neq \emptyset$;
3. $\forall p \in s$, $\exists c \in C_\Pi$ such that $p \in c$.

We define C_Π and S_Π as the set of cores and the set of survivor sets of Π, respectively. $\Box_{2.1}$

The function $r(.)$ and the target degree of reliability R are used at this point only to formalize the idea of a core. In reality, reliability does not need to be expressed as probabilities. For example, consider the following system representation:

Example 2.2 :

- $\Pi = \{\mathrm{ph}_1, \mathrm{ph}_2, \mathrm{pl}_1, \mathrm{pl}_2, \mathrm{pl}_3, \mathrm{pl}_4\}$
- $C_\Pi = \{\{\mathrm{ph}_1, \mathrm{ph}_2, \mathrm{pl}_1\}, \{\mathrm{ph}_1, \mathrm{ph}_2, \mathrm{pl}_2\}, \{\mathrm{ph}_1, \mathrm{ph}_2, \mathrm{pl}_3\}, \{\mathrm{ph}_1, \mathrm{ph}_2, \mathrm{pl}_4\}\}$
- $S_\Pi = \{\{\mathrm{ph}_1\}, \{\mathrm{ph}_2\}, \{\mathrm{pl}_1, \mathrm{pl}_2, \mathrm{pl}_3, \mathrm{pl}_4\}\}$

$\Box_{2.2}$

In this system, ph_1 and ph_2 are very reliable and each of these fail independently of every other $p \in \Pi$. Processes pl_i, for $1 \leq i \leq 4$, however, fail dependently among each other. That is, for every pair of processes pl_i, pl_j, $1 \leq i, j \leq 4$ and $i \neq j$, we have that if pl_i is faulty in some execution of the system, then pl_j is also faulty. Thus, a subset with maximum reliability contains processes ph_1, ph_2, and exactly one process pl_i. Suppose that the maximum reliability achievable for a subset of processes satisfies the intuitive notion of target degree of reliability for this system. We can therefore infer that for each i, $1 \leq i \leq 4$, $\{ph_1, ph_2, pl_i\}$ is a core. The set C_Π of cores is hence as follows:

In the remainder of this paper, we assume that these subsets are provided as part of the system representation. In the following sections, a system is described by a triple $\langle \Pi, C_\Pi, S_\Pi \rangle$, for Π being a set of processes, C_Π being the set of cores of Π, and S_Π being the set of survivor sets of Π. We call henceforth $\langle \Pi, C_\Pi, S_\Pi \rangle$ a system representation.

2.2 Failure Models

We assume two failure models: crash and arbitrary. When discussing failures, one distinguishes channel failures and process failures. In both models considered here, channels are assumed to be reliable. We define a reliable channel as one that satisfies the following properties:

Validity: If $p, q \in \Pi$ are correct processes and p sends a message m to q, then m is eventually delivered;

Integrity: A process $p \in \Pi$ receives a message m if and only if some process $q \in \Pi$ sent it to p. Moreover, p receives m exactly once.

From these channel properties, if a correct process p_i puts a message m in buffer $input_{ij}$ and p_j is also correct, then m is eventually moved to $output_{ij}$. Also, no message in buffer $output_{ij}$ is spontaneously generated, for any pair of processes $p_i, p_j \in \Pi$. If a message is in $output_{ij}$ at some time t, then it was placed at $output_{ij}$ by p_i at some time $t' < t$.

The possibilities for process failures differentiate the models. In the crash model, processes fail by crashing. That is, if a process p is faulty in an execution, then it prematurely stops sending and receiving messages in that execution. Thus, there is a time t after which p stops receiving and sending messages, even though it was supposed to do it according to the algorithm. In contrast to a crashed process, we say that a process is alive at some time t either if it is correct at t or if it has not crashed at any time $t' < t$.

Although a crashed process p_i does not operate properly after time t, p_i does not accomplish incorrect computations. In the arbitrary model, on the other hand, faulty processes behave arbitrarily, and hence this model is strictly weaker than the crash model. Examples of arbitrary behavior are: forging messages, arbitrarily modifying the content of messages, selectively forwarding messages, and changing states without following the protocol specification. It is important to observe that some arbitrary failures are detectable, whereas others are not [10, 11]. For example, the modification of the initial value of a process p_i is not detectable. This is due to the locality of this information. The initial value of p_i is only known by p and consequently it is not possible to verify whether it was modified arbitrarily or not. On the other hand, some failures are detectable and attributable to some process. Suppose the channels are FIFO. If a process p_i sends malformed or out-of-order messages then a correct process p_j receiving those messages is able to detect that p_i is faulty. Note that FIFO channels are easily implemented by a counter, which has its value sent along with every message and is incremented every time a message is sent. Even if a byzantine process p_i changes the value of a channel counter arbitrarily, it is still possible for a correct process p_j to detect p_i as faulty. We assume FIFO channels for our protocol that solves Consensus in a asynchronous systems with byzantine processes. The issue of FIFO channels is hence addressed again in the section 2.4, which discuss asynchronous systems with arbitrary process failures.

2.3 Synchronous Model

The synchronous model imposes bounds on message delay, process speed, and clock drift. These bounds, however, are not necessarily based on absolute time. As in the model of Dolev *et al.* [12], steps of an algorithm are used to define these bounds. Following this model, the timing assumptions for a synchronous system are given by two parameters: $\Phi \geq 1$ and $\Delta \geq 1$. Furthermore, any execution of an algorithm α in such a system satisfies the following properties:

Process synchrony : for any finite subsequence w of consecutive steps, if some process p_i takes $\Phi + 1$ consecutive steps in w, then any process that is still alive at the end of w has taken at least one step in w;

Message synchrony : for any pair of indices k, l, with $l \geq k + \Delta$, if message m is sent to p_i during the k-*th* step, then m is received by the end of the $l - th$ step.

If these properties hold, then an execution can be further organized in rounds, which are defined in terms of steps of processes. In a round, a process p_i executes $n + k$ steps. The first n steps are used by p_i to send real messages, whereas in the last k steps it sends *null* messages. These k last steps are necessary to guarantee that all messages sent to p_i in a round r are received before p_i proceeds to round $r + 1$. The number k of steps is a function of Δ, Φ, n, and r.

The algorithms for synchronous systems described in sections 4 and 6 are round-based. This format facilitates understanding, since it abstracts several details of the system model. The algorithms are also not described in an automaton format, since the description would be longer and would not improve clarity. Instead, we use sequential code to present the algorithms. States and transitions, however, are easily observable from the changes on the values stored by the variables used by the algorithm.

2.4 Asynchronous Model

In an asynchronous system, there is no bound on message delay, process speed, or clock drift [2, 13, 9, 14]. Thus, in such a system, a message sent from a correct process p_i to some other process p_j may take arbitrarily long to be received. Message delay, although considered to be unbounded, is assumed to be finite. This is due to the validity property of the channels, which says that every message sent from a correct process p_i to another correct process p_j is eventually received.

According to the FLP result [15], it is not possible to solve Consensus in a pure asynchronous system, even if only a single crash failure is assumed. The intuition behind the impossibility is that it is not possible to distinguish a crashed process from a very slow one. As discussed previously, a message sent may take a finite but unbounded amount of time to reach its destination, preventing processes from distinguishing some executions from others. It is therefore necessary to assume some liveness property for the system that guarantees that something good will eventually happen and will hold long enough so that correct processes can reach agreement.

Chandra and Toueg proposed to extend the asynchronous model with an oracle that provides information about process failures. This oracle is called a failure detector [2]. Briefly, each process has a failure detector module available to itself, and it queries the module every time the algorithm requires failure information. They showed in their work that failure detectors do not need to detect crash failures perfectly to make Consensus solvable in such extended model. Moreover, they proved that a failure having the properties of $\diamond\mathcal{W}$ is necessary [16]. Another interesting result out of their work is the equivalence between the classes $\diamond\mathcal{W}$ and $\diamond\mathcal{S}$, meaning that given a failure detector $\mathcal{D}$ of one of the classes, there is an algorithm that transforms $\mathcal{D}$ into a failure detector $\mathcal{D}'$ of the other class. In this paper, we assume an asynchronous model with crash process failures extended with a failure detector $\mathcal{D} \in \diamond\mathcal{S}$. The properties that define a failure detector $\mathcal{D} \in \diamond\mathcal{S}$ are as follows:

Strong completeness : Eventually every process that crashes is permanently suspected by every correct process;

Eventual weak accuracy : There is a time after which some correct process is never suspected by any correct process.

In section 5, we assume an asynchronous model extended with a failure detector $\mathcal{D} \in \diamond S$.

For a byzantine setting, other classes of failure detectors are proposed in the literature. Malkhi and Reiter describe the failure detector class $\diamond\mathcal{S}(bz)$ [11]. A failure detector $\mathcal{D}$ in $\diamond\mathcal{S}(bz)$ provides information about quiet processes only. By definition, a quiet process is a faulty process which sends a finite number of messages in an infinite execution. Thus, a failure detector $\mathcal{D}$ is not supposed to detect any other faulty behavior other than silence. The detection of other arbitrary behaviors is implemented by a distributed algorithm. This is illustrated in [11] by an algorithm which relies on the detection of malformed, out-of-order, and unjustifiable messages to solve Consensus, thus showing that the properties of $\diamond\mathcal{S}(bz)$ are sufficient for an asynchronous system with byzantine failures. The definition of $\diamond\mathcal{S}(bz)$, however, assumes a strong system model. It assumes a reliable broadcast primitive, which also satisfies causal order, to exchange messages [13] and authenticated [1], reliable channels between pairs of processes. By assumption, every message is broadcast to all the processes using the given primitive. This prevents that faulty processes send different messages to different processes in a broadcast.

Differently from Malkhi and Reiter, Kihlstrom *et al.* define a class $\diamond S$(Byz) of failure detectors which expose arbitrarily faulty processes. [10] As in the previous definitions, each process has a failure detector module that output a list of processes suspected of having presented detectable arbitrary failures. Note that the definition of detectable arbitrary failures includes omission failures, hence detecting quiet processes as well. The algorithm shown in their work to solve Consensus is tightly coupled to the failure detector, since it has to provide certificates that justify messages sent. The failure detector thus uses these certificates to validate the choices made by the algorithm. Note that this validation mechanism is viable only by assuming the certificates to be unforgeable. An important observation is that the system model assumed is weaker than the model assumed in the definition of $\diamond\mathcal{S}(bz)$. Processes send messages to each other through end-to-end reliable channels, guaranteeing that a message sent from a correct process to another correct process is eventually received.

The last class of failure detectors for arbitrary settings we discuss here is $\diamond M$, proposed by Doudou and Schiper. [17] A failure detector of this class satisfies the mute completeness property, besides the eventual weak accuracy defined previously. The definition of a mute processes resembles the definition of a quiet process, but the former is more comprehensive. An advantage over the $\diamond\mathcal{S}(bz)$ class is again the weaker system model assumed. We now repeat the definitions of a mute process and mute completeness as presented in [17].

Mute process : Let p_i and p_j be two processes. Process p_i is mute to p_j if there is a time after which either (1) p_i crashes, or (2) p_i stops forever sending messages to p_j, or (3) p_i sends only incorrect signed messages (sender cannot be identified) or unsigned messages to p_j.

Mute Completeness : There is a time after which every process p_i, that is mute to a correct process p_j, is suspected forever by p_j.

The failure detector is not tightly coupled to the algorithm that solves Consensus in [17]. Although the failure detector verifies signatures, these are not assumed to be

---

[1] The authentication mechanism is assumed to be unforgeable

generated by the algorithm. Unforgeable signatures are assumed to be available as part of the system model. The only stronger assumption made in terms of the system model compared to the one assumed by Kihlstrom *et al.* is the FIFO property for the communication channels. This property is required by the Consensus algorithm, though, and not by the failure detector. As observed before, the FIFO property for a channel is implemented by a counter, which is incremented every time a message is sent and its current value goes along with every message. Even if a faulty process p_i changes arbitrarily the value of the counter sent with a message to p_j, p_j eventually detects p_i as faulty. If p_i never sends a message with the value expected by p_j, then p_j eventually suspects p_i as mute, by the mute completeness property of the failure detector. On the other hand, if eventually p_i sends a message with the correct counter value, but the message is not the one expected according to the algorithm, then p_i is detected by p_j as a byzantine process. Implementing FIFO channels has its own problems however. One such a problem is the size of the buffer that holds messages received in advance. Implementation details, however, are out of the scope of this work.

Based on the properties of three classes described above, our opinion is that the failure detector as an abstraction should only satisfy enough properties so that it enables the system to overcome the FLP impossibility result. That is, it should provide only the necessary information to enable the system to make progress, guaranteeing liveness. Adding detection of byzantine behavior to the failure detector is a design decision, and does not help in overcoming the impossibility of solving Consensus in an asynchronous model. Moreover, the system model should be as weak as possible, so that it facilitates implementations. We therefore assume in section 7 an asynchronous model extended with a failure detector of the $\diamond M$ class. Out of the three discussed here, $\diamond M$ has the best trade-off in terms of the system model assumptions and failure detector properties. [2]

In sections 5 and 7, we describe algorithms for Consensus in asynchronous systems. Both algorithms simulate rounds asynchronously. Differently from synchronous rounds, asynchronous rounds cannot have their boundaries determined by elapsed time or number of steps, due to the timing assumptions. Typically, a process decide for the end of a round independently from other processes by identifying some pattern of events. For instance, the reception of one message from every process in some particular subset of processes. More details are provided in the sections that describe the algorithms.

2.5 Executions

An execution of an algorithm is essentially a sequence of steps of the processes in Π. There are, however, other details that characterize an execution, such as the initial configuration of the processes, the history of failures of the processes, and the step schedule. These attributes are important, because a difference in one of them may change the result of the computation. For example, the same sequence of steps with a different time schedule may change the decision value in an execution of a Consensus algorithm.

An execution α of an algorithm $\mathcal{A}$ is defined as a tuple $\langle F_\alpha, I_\alpha, S_\alpha, T_\alpha \rangle$. This definition is based on the one by Chandra and Toueg [2] and Charron-Bost *et al.* [14]. $F_\alpha(t)$ evaluates to the subset of processes that have failed by time t. A direct implication of this

---

[2] Ideally, we would choose the weakest failure detector to solve Consensus in a byzantine setting. Kihlstrom *et al.* claim that a failure detector implementing only the properties of $\diamond S(\text{Byz})$ is the weakest failure detector that enables solving Consensus. The $\diamond M$ class, however, is strictly weaker than $\diamond S(\text{Byz})$ and it still enables solving Consensus. Thus, a further analysis on the relations of failure detector classes is necessary, but it is out of the scope of this work, since we are only interested in showing lower bounds for Consensus in our failure model with cores and survivor sets.

definition is that $F_\alpha(t) \subseteq F_\alpha(t+1)$. Because an execution depends on the initial state of the processes, we have that I_α provides the initial configuration of the system. This initial configuration depends on the problem being solved. The Consensus problem, for example, requires every process to have an initial proposed value. Finally, S_α is an infinite sequence of steps of processes in Π. The time t at which a step $e \in S_\alpha$ is executed is given by $T_\alpha(e)$. For every correct process p_i in α, we assume that S_α contains an infinite number of steps of p_i.

Because our asynchronous model is extended with a failure detector, the definition of an execution have to accommodate such feature of the model. First, we revisit the definition of a step. During a step, a process may decide to query its failure detector module. Thus, for asynchronous systems, we add a fourth action to the definition of a step, which is probing its failure detector module for a list of suspected processes. The history of the failure detector in an execution may change the result of the computation and it is henceforth part of the definition of an execution. An execution α of an asynchronous algorithm $\mathcal{A}$ is defined as a tuple $\langle F_\alpha, \mathcal{H}_\alpha, I_\alpha, S_\alpha, T_\alpha \rangle$. The difference from the previous definition is in the inclusion of the failure detector history $\mathcal{H}_\alpha$. The list of processes that p_i suspects at time t is given by $\mathcal{H}_\alpha(i,t)$. Since the failure detector is assumed to be unreliable, the number of suspected processes may increase and decrease as the execution proceeds.

From the definition of an execution, the set of correct process in an execution α is defined as $\mathit{Correct}_\alpha = \Pi - \Cup_{t \in \mathcal{T}_\alpha} F(t)$. The set of failed processes is given by $\mathit{Faulty}_\alpha = \Cup_{t \in \mathcal{T}_\alpha} F(t)$. Note that the mapping $F(t)$ is only useful in the crash failure model. The faulty behavior of a crashed process is observable as soon as it crashes. On the contrary, an arbitrarily faulty process may become faulty at some time t but still behave as a correct process for an unbounded period of time. For this reason, the time by which a process becomes faulty is only considered in the crash failure model. Because we are assuming round-based protocols, we define for the subset of crashed processes that failed by round $r \geq 0$ as $\mathit{Crashed}_\alpha(r)$. A process p_i is in $\mathit{Crashed}_\alpha(r)$ if it has not executed all the steps of some round $r' \leq r$. Neither a correct process nor a faulty process that halts [3] is in $\mathit{Faulty}_\alpha(r)$, for any $r \geq 0$.

3 Consensus

The Consensus problem in a fault-tolerant message-passing distributed system consists, informally, in reaching agreement among a set of processes upon a value. Each process starts with a proposed value and the goal is to have all non-faulty processes deciding on the same value. Throughout the paper, we denote V as the set of possible decision values. Although often a binary set V is sufficient, we assume that V has an arbitrary size to keep the definition as general as possible. Also, we assume that the default value $\perp$ used in the algorithms is not in V. Every time we refer to a value that is either a decision value in V or the default, we use $V \cup \{\perp\}$ to denote all the possibilities.

In the crash failure model, Consensus is often specified in terms of the following three properties [17]:

Validity If some non-faulty process $p_i \in \Pi$ decides on value v, then v is the initial value of some process $p_j \in \Pi$;

---

[3] Some computations are finite, such as distributed Consensus. Thus, we assume that once a correct process halts, it executes an unbounded number of null steps.

Agreement If two non-faulty processes $p_i, p_j \in \Pi$ decide on values v_i and v_j respectively, then $v_i = v_j$;

Termination Every correct process eventually decides.

The validity property as specified above assumes that no process will ever try to cheat on its proposed value. This is true in the crash failure model, but unrealistic assuming arbitrary process failures. Although a byzantine process cannot prevent agreement by cheating on its proposed value, it can prevent progress. For example, assuming that the only possible decision values are either write or abort, with the above validity property, a faulty process may prevent correct processes from writing if they are all ready to do so, and consequently from making progress. Thus, in the arbitrary model, strong validity is usually considered instead ofvalidity [17, 10]. Strong validity is stated as follows:

Strong validity If the proposed value of process p is v, for all $p \in \Pi$, then the only possible decision value is v.

Strong validity only considers the case in which all processes have the same initial value. Intuitively, this is sufficient to prevent a byzantine process from disrupting the normal behavior of a system when all non-faulty processes are enabled to make progress. When the system is facing problems and not all of the processes propose the same value, however, this property allows the decision value to be arbitrary in the set of possible decision values. That is, the decision value v of non-faulty processes can be either the value proposed by a faulty a processes or even a value that was not proposed by any process, assuming the set of decision values is not binary.

An alternative validity property is proposed by Schiper, called vector validity. [17] The vector validity property says that every correct process has to agree on a vector of proposed values, such that the vector has one value for each process in Π. In addition, for every correct process p_i, the value attributed to p_i has to be the initial value of p_i, and the vector has to contain the value of at least $t+1$ correct processes. In the case that every process has to decide on a single value, the decision value is chosen from this vector by some deterministic strategy: majority, minimum value, etc. Even this property cannot prevent processes from deciding upon the value proposed by a faulty process when the initial value is not the same for every process. According to our assumptions, the two properties do not differ, and hence we choose the strong validity property for simplicity.

4 Synchronous Consensus with Crash Failures

Consensus in a synchronous system with crash process failures is solvable for any number of failures. [18] In the case that all processes may fail in some execution before agreement is reached, though, it is often necessary to recover the latest state prior to total failure for recovery purposes. [19] Since we assume that failed processes do not recover, we don't consider total failure in this work. That is, we assume that the following condition holds for a system representation $\langle \Pi, C_\Pi, S_\Pi \rangle$:

Property 4.1 $C_\Pi \neq \emptyset$. $\Box_{4.1}$

Property 4.1 implies that there is at least one correct process in any execution. We now describe a protocol for a synchronous system represented by $\langle \Pi, C_\Pi, S_\Pi \rangle$, assuming that property 4.1 holds for this system. The protocol is based on the early-deciding protocols discussed by Charron-Bost and Schiper [18], Lamport and Fischer [20]. Algorithms that consider the actual number of failures f are important because they reduce the latency on the common case in which just a few process failures occur. An important observation made by Charron-Bost and Schiper [18] is that there is a fundamental difference between early-deciding protocols and early-stopping protocols for Consensus. In an early-deciding protocol, a process may be ready to decide, but may not be ready to halt, whereas an early-stopping protocol is concerned about the round in which a process is ready to halt. One consequence of this difference is that the lower bound on the number of rounds is not the same. For early-stopping algorithms, there is some execution in which a correct process takes at least $\min(t+1, f+2)$ rounds to halt, for $n \geq t+2$, as shown by Dolev *et al.*. [21] On the other hand, for every early-deciding algorithm, there is some execution in which no correct process decides before $f+1$ rounds, as shown by Charron-Bost and Schiper [18]. In both cases, there are algorithms that meet these bounds, thereby showing that they are tight.

We now describe algorithm **SyncCrash** which solves Consensus in a synchronous system with crash process failures, assuming that information about cores and survivor sets is available. Later in this section, we discuss the advantages of considering our model instead of assuming t of n process failures.

The algorithm differentiates the processes of a chosen core $d\text{-}core \in S_\Pi$ and the processes in $\Pi - d\text{-}core$. In a round, every process in *d-core* broadcasts its knowledge of proposed values to all the other processes, whereas processes in $\Pi - d\text{-}core$ listen to these messages. Processes in *d-core* from which a message is not received in a round are known to have crashed, according to the assumptions of the failure model. This observation is used to detect a round in which no process crashed. Processes $p_i \in \Pi$ hence keep track of the processes in *d-core* that crashed in a round, and as soon as p_i detects a round with no crashes p_i decides. As we show later in this section, when such a round r happens, and by assumption it eventually happens, all alive processes are guaranteed to have the same view of the values proposed by the other processes. In other words, all alive processes in r have the same array of proposed values. Once a process p_i in *d-core* decides, it broadcasts a decision message announcing the decision value dec_i it decided upon. All processes receiving this message decide on x_i as well. Thus, only two types of messages are necessary in the protocol: messages containing the array of proposed values and decision messages. Because processes in *d-core* broadcasts at most one message in every round to all the processes in $|\Pi|$, message complexity is given by $O(|d\text{-}core| * |\Pi|)$. Note that the protocols in [18, 20] designed with the t of n assumption have message complexity $O(|\Pi|^2)$. In addition, our algorithm requires $f+1$ rounds for all the processes to decide if $\Pi \neq d\text{-}core$, and $\min(|d\text{-}core|, f+2)$ rounds to halt otherwise, where f is the number of processes in *d-core* that crash in a given execution α. We prove in [22] that these are actually lower bounds on the number of rounds for Consensus in a system represented with our model. By providing a protocol that meet these bounds, we prove them tight.

The idea of using a subset of processes to reach agreement on behalf of the whole set of processes is not new. The Consensus Service proposed by Guerraoui and Schiper utilizes this concept [23]. Their failure model, however, still assumes t of n process failures, and consequently the subset used to reach agreement is not chosen based on information about correlated failures. This is the main point where our work differs.

Before presenting a pseudo-code of the algorithm, we show a table describing the variables used in the protocol. Table 1 describes the variables, and the pseudo-code of **SyncCrash** is presented in figure 1.

$d\text{-}core \in C_\Pi$	Core set chosen as the one responsible for the decision.
$dec_i \in V \cup \{\perp\}$	A process p_i decides once it sets dec_i.
$d \in \{true, false\}$	Boolean variable indicating whether the process decided in the previous round or not.
$pv_i[1 \cdots \|d\text{-}core\|]$, $pv_i[j] \in V$	Vector of proposed values.
$e_i[1 \cdots (\|d\text{-}core\| - 1)]$, $e_i[r] \subset d\text{-}core$	Array of failed processes. $e_i[r]$ stores subset of processes detected by p_i as crashed at round r.

Table 1: Variables used in the algorithm **SyncCrash**

We now present a proof of correctness for **SyncCrash** in the synchronous model with crash failures. Before proving the theorems showing that our algorithm satisfies the three Consensus properties, we prove a few lemmas that are used in the proofs of the theorems. Consider the following definition first.

Definition 4.2 Let α be an execution of **SyncCrash**. We denote $\alpha(ijwk)$ as the value $pv_j[k]$ that process p_i receives in a message from process p_j at round $|jwk|$. $\Box_{4.2}$

Lemma 4.3 *Let α be an execution of* **SyncCrash** *and p_i, p_j be two processes such that $p_i \in \text{d-core}$, $p_j \in \Pi$, $i \neq j$. Let $w \in \text{Pid}^*$ be the shortest sequence of processes such that $\alpha(iwj) = x$, $x \in V$, $x \neq \perp$, assuming such a sequence exists. For every round r, $1 \leq r \leq |iwj| - 1$, the value stored in $\text{pv}_i[j]$ is $\perp$. For every round r, $|iwj| \leq r \leq |\text{d-core}|$, $f = |\text{d-core}| - |(\text{d-core} \cap Correct(\alpha))|$, the value stored in $\text{pv}_i[j]$ is x, and x is the initial value of p_j.*

Proof: We prove this lemma by induction on the length of w. The base case consists of $|w| = 0$. If $|w| = 0$, then, at round 1, process p_i receives a message from process p_j containing its initial value x, and it stores this value in $pv_i[j]$. Observe that every message m_k sent in this round by a process $p_k \neq p_j$ is such that $m_k.pv_k[j] = \perp$, and by the algorithm p_i does not update $pv_i[j]$.

Now assume the lemma is valid for all w', $|w'| \leq |w|$. We prove it for $|w| + 1$. Suppose that process p_i receives a message from process p_k, such that $\alpha(ikwj) = x'$, $x' \in V$. Consequently, from the algorithm, process p_i makes $pv_i[j] = x'$. By the induction hypothesis, we have that $x' = x$, the initial value of p_j. Moreover, for every other process $p_l \in d\text{-}core$, $p_l \neq p_k$, we have that either $pv_l[j] = x$ or $pv_l[j] = \perp$ at the end of round $|kwj|$. $\Box_{4.3}$

From lemma 4.3 we can extract the following corollary.

Corollary 4.4 *Let α be an execution. $\forall p_i \in \text{d-core} \cap \text{Correct}(\alpha), p_j \in \text{Correct}(\alpha)$, $\forall r \in \{1 \cdots |\text{d-core}|\}$, we have that $\text{pv}_j[i] = x$ at the end of round r, for $x \in V$ being the initial value of process pv_i.*

Proof: If $p_i \in d\text{-}core$ is correct, then for every correct process p_j, we have that $\alpha(ji) = x$. From lemma 4.3, for every round r, $r \geq 1$, we have that $pv_j[i] = x$. $\Box_{4.4}$

The next three lemmas form a substantial part of the proof that **SyncCrash** satisfies agreement. The following definition is used in the statement of the three lemmas.

```
Algorithm $SyncCrash$ for process $p_i$:
Input: set $\Pi$ of processes; set $C_\Pi$ of cores; initial value $v_i \in V$

Initialization:
    $d\text{-}core \in C_\Pi$; $dec_i \leftarrow \perp$; $d \leftarrow false$
    $pv_i[1 \cdots |d\text{-}core|]$, $pv_i[k] = \perp$, $\forall k \in [1 \cdots |d\text{-}core|]$, $k \neq i$. If $p_i \in d\text{-}core$, $pv_i[i] \leftarrow v_i$
    $e_i[1 \cdots (|d\text{-}core| - 1)]$, $e_i[k] = d\text{-}core$, $\forall k \in [1 \cdots (|d\text{-}core| - 1)]$

Round $1 \leq r < |d\text{-}core|$, $\forall p_i \in d\text{-}core$:
  if $(d = false)$ then
    send$(i, pv_i)$ to all process in $d\text{-}core$
    send$(i, pv_i)$ to all process in $\Pi - d\text{-}core$
  else
    send$(Decide, dec_i)$ to all processes in $d\text{-}core$
    send$(Decide, dec_i)$ to all processes in $\Pi - d\text{-}core$
    halt
  upon reception of $(m = (Decide, dec_j))$ do
        $dec_i \leftarrow dec_j$
        $d \leftarrow true$
  upon reception of $(m = (j, pv_j))$ do
        $e_i[r] \leftarrow e_i[r] - \{j\}$
        for $k = 1$ to $|\Pi|$ do
            if $(pv_j[k] \neq \perp)$ then $pv_i[k] \leftarrow pv_j[k]$
  if $(((e_i[r-1] = e_i[r]) \wedge (d = false)) \vee (r = |d\text{-}core| - 1))$ then
        $dec_i \leftarrow \min(pv_i[k])$
        $d \leftarrow true$

Round $|d\text{-}core|$, $\forall p_i \in d\text{-}core$:
  send$(Decide, dec_i)$ to all processes in $\Pi - d\text{-}core$
  halt

Round $1 \leq r \leq |d\text{-}core|$, $\forall p_i \in \Pi - d\text{-}core$:
  upon reception of $(m = (Decide, dec_j))$ do
        $dec_i \leftarrow dec_j$
        halt
  upon reception of $(m = (j, pv_j))$ do
        $e_i[r] \leftarrow e_i[r] \cup \{j\}$
        for $k = 1$ to $|\Pi|$ do
            if $(pv_j[k] \neq \perp)$ then $pv_i[k] \leftarrow pv_j[k]$
  if $((e_i[r-1] = e_i[r]))$ then
        $dec_i \leftarrow \min(pv_i[k])$
        halt
```

Figure 1: Synchronous Consensus for Dependent Crash Failures

Definition 4.5 Let:

1. $\alpha = \langle F_\alpha, I_\alpha, S_\alpha, T_\alpha \rangle$ be an execution of **SyncCrash**;
2. p_i, p_j be two processes in $\Pi - Crashed(\alpha, r)$, where r is a round of α;
3. $e_i \in S_\alpha$ be a step of p_i such that p_i receives its last message of round r at step e_i;
4. $e_j \in S_\alpha$ be a step of p_j such that p_j receives its last messages of round r at step e_j;

5. $e'_i, e'_j \in S_\alpha$ be any two steps of p_i and p_j, respectively, at round r, such that $T(e'_i) \geq T(e_i)$ and $T(e'_j) \geq T(e_j)$.

We say that processes p_i and p_j have identical vectors at round r if and only if for every $p_k \in$ *d-core* and, $pv_i[k] = pv_i[k]$, where pv_i is the vector of proposed values of p_i after taking step e'_i and pv_j is the vector of proposed values of p_j after taking step e'_j. $\Box_{4.5}$

Lemma 4.6 *Let α be an execution of* **SyncCrash**. *If r is a round of α in which no process crashes, then for every $p_i, p_j \in (\Pi - \mathrm{Crashed}(\alpha, r))$ p_i and p_j have identical vectors in r.*

Proof: If no process crashes in r, then every process $p_i \in (\Pi - \mathit{Crashed}(\alpha, r))$ receives the same set of messages M. A message $m_j \in M$ contains the vector of proposed values of process p_j. From the algorithm, for every entry $m_j.pv_j[k]$ with a value v, $v \in V$ and $v \neq \perp$, p_i updates $pv_i[k]$ with the same value v. Note that for every entry k, there are no two messages in M indicating distinct values $v, v' \in V$, by Lemma 4.3. Thus, once a processes p_i and p_j receive every message sent to them at round r and update their respective vectors pv_i and pv_j accordingly, we have that $pv_i[k] = pv_j[k]$ for every $k \in \mathit{Pid}$.

An alive process p_k in r decides if it either receives messages from the same subset of processes in both rounds $r-1$ and r, or it receives a decide message. Otherwise, it moves on to round $r+1$ by the end of round r. An important observation is that p_k cannot receive at round r a message from some process p_l from which p_k does not receive a message at round $r-1$. This is due to the assumptions that channels are reliable and processes only fail by crashing.

By assumption, no process crashes in r. Processes p_i and p_j have to receive all the messages sent to them at round r and updating their respective vector of proposed values before either deciding in r or moving to round $r+1$. We conclude that p_i and p_j have identical vectors at r. $\Box_{4.6}$

Lemma 4.7 *Let α be an execution of* **SyncCrash**, *$r > 1$ be a round in which every process in $\Pi - \mathrm{Crashed}(\alpha, r-1)$ has an identical vector of proposed values before receiving any messages in r, and $p_i, p_j \in (\Pi - \mathrm{Crashed}(\alpha, r))$ be two processes that do not receive a decide message at round r. Processes p_i and p_j have identical vector at round r.*

Proof: By assumption, every two processes p_k and p_l that send at least one message in r do so with the same array of proposed values. Thus, even if two alive processes p_i and p_j in r receive different sets of messages, no updates at the vector of proposed values occur in none of the processes. In such a round, for every message m_k an alive process p_i in r receives, we have that $m_k.pv_k = pv_i$, and consequently no entry in pv_i changes its value after p_i receives every delivered message at round r. Process p_i is some arbitrary alive process in r, and hence the previous observation generalizes to every alive process in r.

Because there are no updates in the vector of proposed values of any alive process and by assumption these vectors are the same in the beginning of round r, we have that $pv_i = pv_j$ before deciding at round r' or moving to round $r'+1$. Processes p_i and p_j therefore have identical vectors at round r. $\Box_{4.7}$

Lemma 4.8 *Let α be an execution of* **SyncCrash**, *r be the first round of α in which no process crashes. For every round $r' \geq r$, if p_i and p_j are alive processes at round r', then p_i and p_j have identical vectors at round r'.*

Proof: We prove this lemma with a simple induction on the round numbers. Let the base case be round r. From lemma 4.6, every alive process at round r has the same vector of proposed values before deciding at round r or moving to round $r+1$. Assume now that the proposition is true for every $r' \geq r$. We prove for $r'+1$. By assumption, we have that p_i and p_j have identical vectors at round r', for where $p_i, p_j \in (\Pi - \mathit{Crashed}(\alpha, r'+1))$. Thus, both p_i and p_j begin round $r'+1$ with the same vector of proposed values. From lemma 4.7, p_i and p_j have identical vectors at round $r'+1$. $\Box_{4.8}$

Lemma 4.9 *Let* α *be an execution and* $f = |\text{d-core}| - |(\text{d-core} \cap Correct(\alpha))|$. *For every* $p_i \in \Pi \cap \text{Correct}(\alpha)$, *if* $p_i \in$ d-core, *then* p_i *decides in at most* $\min(|\text{d-core}| - 1, f+1)$, *otherwise* p_i *decides in at most* $f+1$ *rounds.*

Proof: Suppose that f processes in *d-core* fail in execution α, where $0 \leq f < \mathit{d\text{-}core}| - 1$. For every process p_i in $\mathit{Correct}(\alpha)$, p_i decides either when it detects a round without failures or when it receives a decide message. In the former case, p_i cannot detect $f+1$ rounds with failures, because there are f failures by assumption. Thus, it has to decide in some round r, $1 \leq r \leq f+1$. On the other hand, if p_i decides due to the reception of a decide message this cannot happen at a round $r' > (f+1)$, otherwise p_i decides by detecting a round with no failures.

Consider now the special case of $f = |\mathit{d\text{-}core}| - 1$. If a correct process in *d-core* detects $|\mathit{d\text{-}core}| - 1$ rounds with failures and it receives no decide message in a previous round, then it knows at round $|\mathit{d\text{-}core}| - 1$ that every other process in *d-core* has failed. It is safe then to decide and to send a decide message at the last round $|\mathit{d\text{-}core}|$. Note that this is only true because a process in *d-core* sends messages to the other processes in *d-core* first. This implies that no correct process in $\Pi -$ *d-core* knows about more initial values of processes than the correct processes in *d-core*. A consequence of this implication is that a correct process p_j in $\Pi -$ *d-core* cannot do the same in the case it has detected $|\mathit{d\text{-}core}| - 1$ rounds with failures. Process p_j has to wait until round $|\mathit{d\text{-}core}|$ to decide. Thus, a correct process in $\Pi -$ *d-core* again decides in at most $f+1 = |\mathit{d\text{-}core}|$ rounds.

To conclude, let p_i be a process in $\mathit{Correct}(\alpha)$. If $p_i \in$ *d-core*, then it decides in at most $\min(|\mathit{d\text{-}core}| - 1, f+1)$. Otherwise, p_i decides in at most $f+1$ rounds. $\Box_{4.9}$

We now show that **SyncCrash** satisfies the three Consensus properties. Before stating and proving the theorems, we introduce some useful notation. For a given execution α, suppose some process p_i decided upon a value received in a decision message from process p_j. Let $\alpha(w, \mathit{Decide}$, $w \in \mathit{Pid}^*$, be a sequence of processes such that a process p_k in w decides upon the value it receives in a decision message from the process p_l that precedes p_k in w. The only exception is the rightmost process in w, which decides dues to the detection of a round without failures. For example, suppose p_i decides upon the value it receives from p_j in a decision message, p_j decides upon the value it receives from p_k, and p_k is the first process to generate a decision message. With our notation, this is expressed as $\alpha(ijk, \mathit{Decide}$.

Theorem 4.10 *Let* α *be an execution of* **SyncCrash**. **SyncCrash** *satisfies Validity in* α.

Proof: From the algorithm, every correct process in Π decides either when it detects a round without crashes or when it receives a decision message. If a process decides in a given execution α because it detected a round r without crashes, then it decides on the first value of the array that is different from $\perp$. By assumption, there is at least one correct process p_i in *d-core* in any execution α. From corollary 4.4, $pv_j[i]$ has the initial value of p_i, for every correct process $p_j \in Correct(\alpha)$. Thus, there is no execution such that a correct process decides on $\perp$. It remains to show that if a correct process p_i decides on the value $pv_i[k]$, then $pv_i[k]$ contains the initial value of p_k even if p_k is faulty. From lemma 4.3, $pv_i[k]$ is either $\perp$ or the initial value of p_k. According to the algorithm, no process decides on the value $\perp$, consequently, $pv_i[k]$ has to be the initial value of p_k.

In the second case, a process p_i decides when it receives a decision message with a decision value dec_j from some process $p_j \in$ *d-core*. Thus, we assume there is a chain of decide messages $\alpha(ijw, Decide)$, where: 1) $w \in Pid^*$; 2) $i, j \in Pid$. In the suffix jw, let k be the id of the first process that sends a decide message. Because p_k is the first process in the chain, it does not decide upon a value received in a decide message. Process p_k decides because it detects a round without failures. From the first case, p_k decides in a value $v \in V$ proposed by some process in *d-core*. As the value dec_k is forwarded along the chain, every process in ijw decides on dec_k. Process p_i therefore decides upon dec_k as well. We conclude that validity is satisfied. $\Box_{4.10}$

Theorem 4.11 *Let α be an execution of* **SyncCrash**. **SyncCrash** *satisfies Agreement in α.*

Proof: Let r be the earliest round in which some process $p_i \in \Pi$ decides in α. By the algorithm, if p_i decides in r, then p_i receives messages from the same subset of processes in both rounds $r-1$ and r. From the assumptions for the failure model, we have that no process crashed either in round r or in round $r-1$. By Lemma 4.8, for every round $r' \geq r$ and $p_j, p_k \in \Pi - Crashed(\alpha, r')$, we have that p_j and p_k have identical vectors.

If any process $p_j \in \Pi$ decides in a round $r' \geq r$, then either p_j detects that there was no failure at the previous round or p_j receives a decision message from some other process $p_k \in$ *d-core* $- Crashed(\alpha, r'-1)$. In the former case, process p_j decides on the same value as p_i, because $pv_i = pv_j$ and the strategy to choose the decision value from the array is deterministic.

If p_j decides upon the value dec_k received in a decision message from some process $p_k \in$ *d-core*, then there is a chain of decide messages $\alpha(jkw, Decide)$, where $w \in Pid^*$, and $j, k \in Pid$. In the suffix jkw, let l be the id of the first process that sends a decide message. Note that l can be either k or the id of some other process. Because p_l is the first process in the chain, it does not decide upon a value received in a decide message. Process p_l decides because it detects a round without failures. From the first case, p_l decides upon the same value as p_i. As the value dec_l is forwarded along the chain, every process in jkw decides on dec_l. Thus, p_j decides upon dec_l, which is the same value as dec_i. We conclude that agreement holds in α. $\Box_{4.11}$

Theorem 4.12 *Let α be an execution of* **SyncCrash**. **SyncCrash** *satisfies Termination in α.*

Proof: From lemma 4.9, every correct process eventually decides. $\Box_{4.12}$

By characterizing correlated process failures with cores and survivor sets, we improve performance both in terms of message and time complexity. For example, consider again the six process system described in Example 2.2. By assuming t of n failures, t must be as large as the maximum number of failures among all valid executions, which is five. Thus, it is necessary to have at least five rounds to solve Consensus in the worst case.

By executing **SyncCrash** with a minimum-sized core as *d-core*, only three rounds are necessary in the worst case. In addition, no messages are broadcast by the processes in Π − *d-core*. This is different from most protocols designed under the t of n assumption [20, 18, 21], although the same idea can be applied by having only a specific subset of $t+1$ processes broadcasting messages.

5 Asynchronous Consensus with Crash Failures

Given a system representation $\langle \Pi, C_\Pi, S_\Pi \rangle$, suppose the following properties for this system:

Property 5.1 (**Crash Partition**) Any partition (A, B) of Π is such that either A or B contain a core. $\square_{5.1}$

Property 5.2 (**Crash Intersection**) S_Π forms a coterie. $\square_{5.2}$

Claim 5.3 *Crash Partition* $\equiv$ *Crash Intersection.*

Proof:

- Crash Partition → Crash Intersection

  We need to prove that the following properties hold:

  5.3.1: If $s_1, s_2 \in S_\Pi$, then $s_1 \cap s_2 \neq \emptyset$;

  5.3.2: There are no $s_1, s_2 \in S_\Pi$ such that $s_1 \subset s_2$.

  First, we prove 5.3.1 by contradiction. Assume a system configuration in which Crash Partition holds and there are two survivor sets $s_i, s_j \in S_\Pi$ such that $s_i \cap s_j = \emptyset$. In any partition (A, B), either A or B contain elements from all survivor sets. Now suppose the following partition (A, B): $A = s_1$, and $B = (\cup_{s_i \in S - \{s_1\}} s_i)$. In this partition, neither A nor B contain elements from all survivor sets. Consequently, neither of them contains a core, contradicting our assumption that property 5.1 holds.

  The proof for property 5.3.2 follows directly from the definition of survivor sets. Survivor sets are minimal by construction.

- Crash Intersection → Crash Partition

  We prove by contradiction. Assume a system configuration in which Crash Intersection holds and there is a partition (A, B) of Π such that none of A and B contains a core. For every pair of survivor sets $s_1, s_2 \in S_\Pi$, we have that $s_1 \cap s_2 \neq \emptyset$. In order to construct a partition (A, B) such that there is no core in none of the subsets, these properties have to hold for both A and B:

  5.3.3: For every $s_i \in S_\Pi$, we have that $s_i \not\subseteq A$ and $s_i \not\subseteq B$;

  5.3.4: There exist survivor sets $s_i, s_j \in S$, $s_i \neq s_j$, such that $A \cap s_i = \emptyset$ and $B \cap s_j = \emptyset$.

  By showing that both cannot be satisfied at the same time, we reach our contradiction. If we construct a partition (A, B) of Π such that this partition satisfy 5.3.3, then both A and B contain at least one element of every survivor set $s_i \in S_\Pi$ and consequently both A and B contain cores. On the other hand, if we construct a partition (A, B) that satisfy 5.3.4, then we have that $s_i \subseteq B$. In this case, B contains a core. Thus, 1 and 2 cannot be satisfied at the same time by any partition. Consequently, any partition (A, B) is such that either A or B contains a core.

$\Box_{5.3}$

5.1 Lower bound on process replication

Chandra an Toueg showed that $n > 2t$, for n being the number of process and t the maximum number of crashed processes in any execution, is the lower bound on process replication for solving Consensus in an asynchronous system extended with a failure detector of the class $\diamond S$ [2]. This lower bound assumes independent and identically distributed process failures. In our failure model, the **Crash Intersection** (**Crash Partition**) property happens to be the generalization of the $n > 2t$ lower bound. The proof idea is similar to the one used by Chandra and Toueg.

Assume there is an algorithm $\mathcal{A}$ that solves Consensus in some system *sys*$= \langle \Pi, C_\Pi, S_\Pi \rangle$. In addition, suppose that there is a partition (A, B) of the processes in Π such that neither A nor B contains a core. Thus, we build an execution in which the agreement property is violated, no matter what the algorithm does. We build two preliminary executions, α and β, in the process of building an execution γ that violates agreement. For execution α of $\mathcal{A}$, suppose that all the processes in A are correct and the processes in B crash before sending a single message. From the termination property, every process in A eventually decides, and they all have to decide upon the same value v in order to satisfy agreement. Suppose that all the processes in A have the same initial value v_a. By the validity property, we have that $v = v_a$.

The execution β is analogous to α. For β, however, all the processes in B are correct and all the processes in A crash before sending a single message. We assume also, that all the processes in B have the same initial value v_b, and $v_b \neq v_a$. Again from the three Consensus properties, every correct process $p_i \in B$ eventually decides, and p_b decides upon v_b.

Now suppose an execution in which every process in Π is correct. We describe an execution γ that looks the same as α for the processes in A, and the same as β for the processes in B. In γ, the initial value for every process in A is v_a and for every process in B is v_b. Let t_a be the time by which all processes in A have decided in α, and t_b the time by which all processes in B have decided in β. We use t_a and t_b to define message schedule and failure detector history. The messages sent among process in A are scheduled as in α, whereas the messages among processes in B are scheduled as in β. The messages from processes in A to processes in B, and from processes in B to processes in A are only delivered after time $t > \max(t_a, t_b)$. The failure detector history follows the same pattern. For the processes in A, the failure detector history is the same as in α up to time t_a. Processes in B have the same history as in β up to time t_b.

Considering the previous definitions for executions α, β, and γ, processes in A and processes in B cannot distinguish executions α and β, respectively, from execution γ. Hence, processes in A decide v_a, albeit processes in B decide v_b. Execution γ therefore violates agreement independently of what algorithm $\mathcal{A}$ does.

We now prove our proposition more formally.

Theorem 5.4 *Let an asynchronous system* sys *extended with a failure detector of the class $\diamond S$ be represented by $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system. If Consensus is solvable in* sys, *then* sys *satisfies the* **crash partition** *property.*

Proof: We prove this theorem by contradiction. Assume that there is an algorithm $\mathcal{A}$ that solves Consensus in *sys*, albeit *sys* does not satisfy the **crash partition** property. That is, there is at least one partition (A, B) of the processes in Π, such that none of A or B contains a core. We show that there is an execution γ in which the agreement property is violated.

We define first two other executions, α and β, which are used to build γ. Let $\alpha = \langle F_\alpha, \mathcal{H}_\alpha, I_\alpha, \mathcal{S}_\alpha, T_\alpha \rangle$ be as follows:

$$\begin{aligned} F_\alpha(t) &= B, \forall t \geq 0 \\ \mathcal{H}_\alpha(t,i) &= B, \forall t \geq 0, \forall p_i \in A \\ I_\alpha(i) &= v_\alpha, v_\alpha \in V, \forall i \in \Pi \end{aligned}$$

The sequence of steps $\mathcal{S}_\alpha$ and timestamps T_α are dependent on the algorithm, and hence we do not specify them in order to keep the definition compliant with any possible algorithm. The only assumption we make is that there is a finite time t_a such that for every $p_i \in \mathit{Correct}(\alpha)$, there is a step $e \in \mathcal{S}_\alpha$ of p_i in which p_i decides, $T_\alpha(e) \leq t_a$. By assumption, algorithm $\mathcal{A}$ solves Consensus and therefore it has to satisfy the termination property. Thus, such a t_a has to exist.

Now let $\beta = \langle F_\beta, \mathcal{H}_\beta, I_\beta, \mathcal{S}_\beta, T_\beta \rangle$ be as follows:

$$\begin{aligned} F_\beta(t) &= A, \forall t \geq 0 \\ \mathcal{H}_\beta(t,i) &= A, \forall t \geq 0, \forall p_i \in B \\ I_\beta(i) &= v_\beta, \forall i \in \Pi, v_\beta \in V, v_\beta \neq v_\alpha \end{aligned}$$

By the same argument presented before, we do not define $\mathcal{S}_\beta$ and T_β, although we assume that there is a time t_b such that, for every $p_i \in \mathit{Correct}(\beta)$, there is a step $e \in \mathcal{S}_\beta$ of p_i in $\mathcal{S}_\beta$ in which p_i decides, $T_\beta(e) \leq t_b$.

$$\begin{aligned} F_\gamma(t) &= \emptyset, \forall t \geq 0 \\ \mathcal{H}_\gamma(t,i) &= \begin{cases} \mathcal{H}_\beta(t,i) & \forall t \leq t', \forall p_i \in B \\ \mathcal{H}_\alpha(t,i) & \forall t \leq t', \forall p_i \in A \\ \emptyset & \forall t > t', \forall p_i \in \Pi \end{cases} \\ I_\gamma(i) &= \begin{cases} v_\alpha & , \forall p_i \in A \\ v_\beta & , \forall p_i \in B \end{cases} \end{aligned}$$

$\mathcal{S}_\gamma$ and T_γ are defined algorithmically as follows:

- For every $e_a \in \mathcal{S}_\alpha$ such that $T_\alpha(e_a) < \max(t_a, t_b)$, we have that $e_a \in S_\gamma$ and $T_\gamma(e_a) = T_\beta(e_a)$;
- For every $e_b \in \mathcal{S}_\beta$ such that $T_\beta(e_b) < \max(t_a, t_b)$, we have that $e_b \in S_\gamma$ and $T_\gamma(e_b) = T_\beta(e_b)$;
- If $e \in S_\gamma$ and $T_\gamma < \max(t_a, t_b)$, then either $e \in \mathcal{S}_\alpha$ or $e \in \mathcal{S}_\alpha$. If $e \in \mathcal{S}_\alpha$, then $T_\alpha(e) < \max(t_a, t_b)$, otherwise $T_\beta(e) < \max(t_a, t_b)$;

- Let $e \in S_\gamma$ be a step in which a process $p_i \in A$ receives a message from a process $p_j \in B$. We have that for every such a step, $T_\gamma(e) > \max(t_a, t_b)$;
- Let $e \in S_\gamma$ be a step in which a process $p_i \in B$ receives a message from a process $p_j \in A$. We have that for every such a step, $T_\gamma(e) > \max(t_a, t_b)$;

A process $p_i \in A$ cannot distinguish execution α from execution γ, whereas process $p_j \in B$ cannot distinguish execution β from execution. Thus, p_i and p_j have to decide upon v_a and v_b, respectively, therefore violating the agreement property of Consensus.

$\square_{5.4}$

5.2 An algorithm to solve Consensus

As discussed before, Consensus is not solvable in a pure asynchronous system. An approach to overcome this impossibility is to extend the asynchronous model with a failure detector. Here we assume a failure detector $\mathcal{D}$ of the class $\diamond S$, which satisfies the strong completeness and eventual weak accuracy properties. The algorithm we describe uses this failure detector to guarantee liveness.

As the algorithm proposed by Chandra and Toueg [2], our algorithm **AsyncCrash** is based on the rotating coordinator paradigm and proceeds in asynchronous rounds. In every asynchronous round, one process is chosen as the coordinator of that round. The knowledge of which process is the coordinator of some round is pre-determined, and hence there is no need to use leader-election algorithms or similar approaches. The coordinator of a round is responsible for gathering the estimates of some survivor set $S \in S_\Pi$ and for choosing a value out of the ones received from the processes in this survivor set. In the algorithm, the coordinator chooses the value from the process that updated it in the latest round among all the estimates received from the processes in S. Once the coordinator chooses a value, it sends a message to informed all the processes of its estimate. A process that receives this message from the coordinator echoes the coordinator estimate to all the other processes. A process decides as soon as it receives an echo from all the processes in some survivor set $S' \in S_\Pi$, not necessarily the same as S.

So far, we assumed that the coordinator is correct. If the coordinator crashes and no correct process receives an estimate from the coordinator, then eventually all the processes in some survivor set containing only correct processes suspect that the coordinator crashed. This is guaranteed by the strong completeness property of the failure detector. Once a process p_i suspects that the coordinator of its current round has failed, p_i sends a message to all the other processes suggesting the others to move on to the next round. If a process receives a message to move on from all the processes in some survivor set, then it re-initializes its variables and moves on to the next round.

The use of echo messages is not really necessary, but it may anticipate decision when the coordinator c_r of round r crashes at r and at least one correct process, say p_i, receives either a message from the coordinator or an echo message from some other process p_j. The echo messages from p_i induce other processes to send echo messages as well, and eventually non-crashed processes executing round r decide. Without the echo messages, every non-crashed processes would need to wait until all the processes in some survivor set containing only correct processes suspect the coordinator and send moveon messages. Furthermore, decision would be postponed, thereby delaying termination. Because the time to suspect the coordinator may be arbitrarily long, this mechanism prevents unnecessary wait in making a decision. Therefore, the argument in favor of echo messages is not correctness, since it is not hard to modify the algorithm to work without it. Its use, however, may reduce the latency in reaching agreement among the correct processes in a real implementation. Schiper proposed originally the utilization of echo messages as an optimization to have a coordinator-based algorithm less dependent on the coordinator in an asynchronous round [17, 24].

Figure 2 shows the pseudo-code of **AsyncCrash**. Every process executes the same algorithm in a run of the system, although processes have different roles in a round. The algorithm is structured in stages, and every process initiates an execution at stage *StartRound*. In the first round, round 0, p_0 is the coordinator. After sending an **Estimate** message to itself, it changes stages, from *StartRound* to *WaitForEstimates*. Once it receives an **Estimate** message from every process in some survivor set, then it sends a **CoordEstimate** message with its proposed value to all the processes. After sending **CoordEstimate** messages, the coordinator changes to stage *Echoes* and behaves as the other processes for the rest of this round. All the other processes go to stage *Echoes* right after sending an **Estimate** message at stage *StartRound*. At stage *Echoes*, every non-crashed process waits for either an **Echo** message or a **MoveOn** message from all the processes in some survivor set $S \in S_\Pi$. By receiving **Echo** messages from the processes in S, a process p_i decides, whereas it moves to stage *GoToNextRound* upon reception of **MoveOn** messages from the processes in S. At the *GoToNextRound* stage, no messages are involved. A process only re-initializes the variables, assigns a new coordinator, and moves on the next round by changing back to stage *StartRound*. This cyclic process continues until all the correct processes eventually decide.

Stage	Indicates the stage the process is in the current round.
Echoes	Set with **Echo** messages received in the current round.
Estimate	Current estimate of process p_i.
EstUpdate	Round in which *Estimate* is updated.
CurEstimates	Set with the **Estimate** messages received by the coordinator.
r	Keeps track of the current round.

Table 2: Variables used in the algorithm **AsyncCrash**

We now provide a proof of correctness for the algorithm **AsyncCrash**. Before stating and proving the theorems that actually show that **AsyncCrash** satisfy the three Consensus properties, we show some preliminary lemmas. The theorems then are easily shown from these lemmas.

Lemma 5.5 *Let α be an execution of* **AsyncCrash** *and p_i be some correct process that does not decide at round r, $r \geq 0$. Eventually p_i moves on to round $r+1$.*

Proof: If a process p_i does not decide at round r, then it neither receives a **Decide** message nor receives an **Echo** message from all processes in some survivor set. If p_i does not receive a **Decide** message, then there is no chain of **Decide** messages $(iwj)_{Decide} \in C\text{-}Decide(\alpha)$, $j \in Pid$, $w \in Pid^*$, such that p_j received an **Echo** message from all processes in some survivor set.

By assumption, at least one survivor set $S \in S_\Pi$ contains only correct processes, and every message sent by a correct process to another process is eventually received. According to the algorithm, the processes in S send an **Echo** message upon reception of either the first **Echo** message or a **CoordEstimate** message. If none of these messages is received by any of the processes in S, then the coordinator is faulty. Eventually the elements of S suspect the coordinator and send **MoveOn** messages. The eventual suspicion of the

```
Algorithm AsyncCrash for process $i$:
Input: set $\Pi$ of processes; set $C_\Pi$ of cores; set $S_\Pi$ of survivor sets; initial value $v_i \in V$

Variables: $Stage \leftarrow StartRound$; $Echoes \leftarrow \emptyset$; $CurEstimates \leftarrow \emptyset$; $Estimate \leftarrow v_i$;
           $EstUpdate \leftarrow 0$; $r \leftarrow 0$

Stages: $StartRound$; $DecisionTentative$; $GoToNextRound$;

Transition function:
  When ($Stage = StartRound$)
      $Send$(Estimate, $i$, $r$, $Estimate$, $EstUpdate$) to the coordinator $p_{c_i}$
      if($c_i = i$) then $Stage \leftarrow WaitForEstimates$
      else $Stage \leftarrow WaitForCoordEstimate$

  When ($Stage = DecisionTentative$)
      upon reception of (Estimate, $j$, $r$ , $v_j$, $r_j$)
        $CurEstimates \leftarrow CurEstimates \cup \{(v_j, r_j)\}$
        if($\exists S \in S_\Pi$ such that $\forall p_k \in S$, (Estimate, $k$, $r$ , $v_k$, $r_k$) $\in CurEstimates$)
        then $r_k \leftarrow \max(r_x | (v_x, r_x) \in CurEstimates)$
               $Estimate \leftarrow v_k$, $(v_k, r_k) \in CurEstimates$; $EstUpdate \leftarrow r$
               $Send$(CoordEstimate, $i$, $r$, $Estimate.v$) to all processes in $\Pi$
               $Stage \leftarrow Echoes$
      upon reception of (CoordEstimate, $j$, $r$ , $v_j$)
        if($Echoes = \emptyset$) then
          $Send$(Echo, $j$, $r$, $v_j$) to all processes in $\Pi$
          $Estimate \leftarrow v_j$; $EstUpdate \leftarrow r$
      upon reception of (Echo, $j$, $r$, $v_j$)
        if ($Echoes = \emptyset$) then
          $Send$(Echo, $j$, $r$, $v_j$) to all processes in $\Pi$
          $Estimate \leftarrow (v_j, r)$
        $Echoes \leftarrow Echoes \cup$ (Echo, $j$, $r$, $v_j$)
        if($\exists S \in S_\Pi$ such that $\forall p_k \in S$, (Echo, $k$, $r$ , $v$) $\in Echoes$, $v \in V$) then
          Decide upon value $v$
          $Send$(Decide, $i$, $v$) to all processes in $\Pi$
          halt
      upon suspicion of $c_i$
        $Send$(MoveOn, $j$, $r$) to all processes in $\Pi$
      upon reception of (MoveOn, $j$, $r$)
        $MoveOn \leftarrow MoveOn \cup$ (MoveOn, $j$, $r$)
        if ($\exists S \in S_\Pi$ such that $\forall p_k \in S$, (MoveOn, $k$, $r$ , $v$) $\in Echoes$, $v \in V$) then
          $Stage \leftarrow GoToNextRound$

  When ($Stage = GoToNextRound$)
      $r \leftarrow r + 1$; $c_i \leftarrow (c_i + 1) \bmod |\Pi|$
      $Echoes \leftarrow \emptyset$; $MoveOn \leftarrow \emptyset$
      $Stage \leftarrow StartRound$

  When ($Stage = *$)
      upon reception of (Decide, $j$, $v$)
        Decide upon value $v$
        $Send$(Decide, $i$, $v$) to all processes in $\Pi$
        halt
```

Figure 2: Asynchronous Consensus with Crash Failures

coordinator by all the processes in S is guaranteed to happen by the strong completeness property of the failure detector. Once process p_i receives a **MoveOn** message from every process $p_j \in S$, p_i moves to stage *GoToNextRound* and proceeds to round $r+1$. $\Box_{5.5}$

Lemma 5.6 *Let α be an execution of* **AsyncCrash** *and r be the first asynchronous round in which some correct process p_i decides. If p_i decides upon value v, then for every asynchronous round $r' > r$, v is the estimate value proposed by the coordinator of r'.*

Proof: We prove this lemma by induction on the round numbers. Initially, we prove for $r' = r+1$, and then for $r'+1$, assuming the lemma is true for r'.

Let $r' = r+1$. By assumption, we have that some correct process p_i decides at round r. If p_i decides at round r upon value v, then it receives one **Echo** message from every process in some survivor set $S \in \mathcal{S}_\Pi$. An alive process p_j sends an **Echo** message to all the processes, including itself, upon reception of either a **CoordEstimate** or an **Echo** message for the first time from some other process. Moreover, p_j updates its estimate upon reception of the first **Echo** message. Because p_j does not crash at round $r+1$ by assumption, if it sends an **Echo** message, then it eventually updates its estimate. From lemma 5.5, every correct process that does not decide at round r eventually moves on to round $r+1$. At the beginning of round $r+1$, the coordinator of that round waits for the estimate of all the processes in some survivor set $S' \in \mathcal{S}_\Pi$. Upon reception of all the **Estimate** messages sent by processes in S', the coordinator chooses the estimate generated at the latest round. By the intersection property assumed for $\mathcal{S}_\Pi$, there is at least one process $p_j \in S'$ such that p_j's estimate is v and it is updated at round r. Consequently, the coordinator of $r+1$ chooses v as its estimate.

Now, assume that the proposition is true for every $r'' \leq r'$. We prove the proposition for $r'+1$. From the inductive assumption, the coordinator of round r' proposes v as its estimate for round r'. Note that the choice of the value v by the coordinator as its estimate for round r' has to be independent of the subset of processes from which it received **Estimate** messages from. In other words, any survivor set containing processes that have not crashed at asynchronous round r' must be capable of inducing the coordinator to choose v as its estimate for that round. We now show that the coordinator $c_{r'+1}$ of round $r'+1$ has to choose v as its estimate for this round. There are two cases to be analyzed. First, suppose that $c_{r'+1}$ receives **Estimate** messages from a survivor set $S \in \mathcal{S}_\Pi$ which contains no processes that updated their estimates in the previous round. From the inductive assumption, $c_{r'+1}$ has to choose v as the coordinator estimate for this round. For the second case, let $S \in \mathcal{S}_\Pi$ be the survivor set from which $c_{r'+1}$ received **Estimate** messages before choosing the coordinator estimate value for round $r'+1$. Suppose that at least one process p_j updated its estimate in the previous round r'. This value has to be v, by the inductive assumption. From the algorithm, $c_{r'+1}$ has to choose the estimate updated at the latest round, and consequently the coordinator estimate for round $r'+1$ has to be v. $\Box_{5.6}$

Lemma 5.7 *Let α be an execution of* **AsyncCrash** *and p_i be some correct process that decides at round r. Process p_i decides upon the value $v \in V$ proposed by the coordinator of round r.*

Proof: A process decides either when it receives an **Echo** message from every process in some survivor set $S \in \mathcal{S}_\Pi$ or when it receives a **Decide** message from some other process. If p_i receives one **Echo** message from every process p_j in some survivor set $S \in \mathcal{S}_\Pi$, then for all $p_j \in S$ there is a chain of **Echo** messages $(jwk)_{Echo} \in \mathit{C\text{-}Echo}(\alpha)$, $j, k \in \mathit{Pid}$, $w \in \mathit{Pid}^*$, such that p_k received a **CoordEstimate** from c_r. Thus, every **Echo** message p_i receives contains the value proposed by the coordinator c_r.

If p_i receives a **Decide** message, then there is a chain of **Decide** messages $(iwj)_{Decide} \in \mathit{C\text{-}Decide}(\alpha)$, $i, j \in \mathit{Pid}$, $w \in \mathit{Pid}^*$, such that p_j received an **Echo** message from all processes in some survivor set. Two cases are possible: the **Decide** message is sent in some previous round $r' > r$ or the **Decide** message is generated by some process at round r. Suppose the former case first. According to lemma 5.6, once some process decides upon a value v' at some round $r' < r$, the value proposed by the coordinator of round $r \geq r'$ has to be v'. Therefore, in this case, p_i decides upon the value proposed by c_r. In the second case, the **Decide** message is generated at this round. Thus, p_j received **Echo** messages from all the processes in some survivor set, and, from the argument above, p_j decides on the value proposed by the coordinator c_r. $\Box_{5.7}$

Lemma 5.8 *Let α be an execution of* **AsyncCrash**. *For every process p_i, if p_i updates its estimate at asynchronous round r, then it does so with the initial value of some process $p_j \in \Pi$.*

Proof: We prove this lemma with an induction on the asynchronous round numbers. For the base case, suppose $r = 0$. From the algorithm, there are two ways for a process p_j to change its estimate. First, if $j = 0$ (p_j is the coordinator), then it receives an **Estimate** message from every process in some survivor set $S \in S_\Pi$. Because this is the first round, all the **Estimate** messages contain the initial values. More specifically, if process p_k is not crashed at round 0 and it sends an **Estimate** message, then this message contains the initial value of p_k. Thus, the coordinator p_0 chooses arbitrarily among the **Estimate** messages, since they are all tagged with round number 0, and updates its estimate variable accordingly. For the second case, p_j is not the coordinator. If p_j does not receive a single **Echo** message, then it proceeds without updating its estimate. The estimate continues hence to be its initial value v_j. On the other hand, if p_j receives at least one **Echo** message, then it updates its estimate. On the other hand, if p_j receives an **Echo** message from some process p_k first, then it updates with the value v_k sent in the **Echo** message. Since p_k sends an **Echo** message at round 0 by assumption, there is a chain of messages $(kwl)_{Echo} \in \mathit{C\text{-}Echo}(\alpha)$, $w \in \mathit{Pid}^*$, $k, l \in \mathit{Pid}$, such that p_l sent the first **Echo** message of this chain. According to the algorithm, p_k received a **CoordEstimate** with the estimate of the coordinator p_j, and consequently all the messages in this chain contain the estimate of the coordinator. The estimate of the coordinator at round 0 is the initial value of some process as we showed before.

Now assume that the proposition is true for every round $r' \leq r$. We prove for asynchronous round $r + 1$. Suppose p_i is the coordinator of round r. Process p_i then updates its estimate based on the values received in the **Estimate** messages sent by every process in some survivor set $S \in S_\Pi$. Observe that every process p_j in S has as its estimate the initial value of some process. For every $p_j \in S$, if p_j has not updated its estimate in any previous round, then its estimate is still v_j. Otherwise, from the inductive assumption, p_j has as its estimate the initial value of some process $p_k \in \Pi$. Consequently, p_i updates its estimate with the initial value of some process. In the case p_i is not the coordinator, it updates its estimate if and only if it receives at least one **Echo** message. If p_i receives a **Echo** message from some other process p_k, then there is a chain $(kwl)_{Echo} \in \mathit{C\text{-}Echo}(\alpha)$, $w \in \mathit{Pid}^*$, $k, l \in \mathit{Pid}$, such that p_l sends the first **Echo** message. According to the algorithm, p_l receives a **CoordEstimate** and sends the **Echo** messages with the estimate of the coordinator. As we showed before, the estimate of the coordinator is the initial value of some process $p_j \in \Pi$. $\Box_{5.8}$

Lemma 5.9 *Let* α *be an execution of* **AsyncCrash**. *Every* $p_i \in \mathrm{Correct}(\alpha)$ *eventually decides in* α.

Proof: From lemma 5.5, every correct process that does not decide in a round r, $r \geq 0$, moves on to the next round. A process moves on by receiving one **MoveOn** message from every process p_j in some survivor set $S \in S_\Pi$. According to the algorithm, a process sends a **MoveOn** message to all the other processes when it detects that the coordinator c_r has failed. From the eventual weak accuracy property of the failure detector, however, there is a time t after which there is some correct process p_k that is permanently not suspected by any other correct process. Therefore, there is time $t' > t$ that p_k becomes the coordinator of some asynchronous round r' and no correct process suspects p_k. No correct process then sends a **MoveOn** message at this round, and consequently no correct process moves on to the next round. Eventually, every correct process receives either an **Echo** message from every process in some survivor set or a **Decide** message and finally decides. $\Box_{5.9}$

We now show three theorems to conclude our proof that **AsyncCrash** solves Consensus in the asynchronous model with crash process failures. In order to accomplish this, we present three theorems, each one showing that one of the Consensus property is satisfied by **AsyncCrash** in every possible execution α.

Theorem 5.10 *Let* α *be an execution of* **AsyncCrash**. **AsyncCrash** *satisfies Validity in* α.

Proof: From lemma 5.7, every correct process that decides at round r decides upon the value v proposed by the coordinator. Before sending a **CoordEstimate** message, the coordinator updates its estimate with v. By lemma 5.8, v has to be the initial value of some process $p_j \in \Pi$. $\Box_{5.10}$

Theorem 5.11 *Let* α *be an execution of* **AsyncCrash**. **AsyncCrash** *satisfies Agreement in* α.

Proof: If $\mathit{Correct}(\alpha)$ contains only one process, then agreement is trivially satisfied. Thus, suppose $\mathit{Correct}(\alpha)$ contains at least two processes. From lemma 5.9, every correct process eventually decides. Let $p_i, p_j \in \mathit{Correct}(\alpha)$, $p_i \neq p_j$, decide at round r_i and r_j respectively. If $r_i = r_j$, then both decide upon the value v proposed by the coordinator of round $r = r_i = r_j$, by lemma 5.7. In the case that $r_i \neq r_j$, they also have to decide upon the same value. Assume without loss of generality that $r_i < r_j$. From lemma 5.7, p_i decide upon the value v proposed by the coordinator, and from lemma 5.6, the coordinator of r_j has to update its estimate with the value v and propose v in the **CoordEstimate** messages it sends. Again from lemma 5.7, if p_j decides at round r_j, then it decides on v. $\Box_{5.11}$

Theorem 5.12 *Let* α *be an execution of* **AsyncCrash**. **AsyncCrash** *satisfies Termination in* α.

Proof: This result follows directly from lemma 5.9. $\Box_{5.12}$

6 Synchronous Consensus with byzantine failures

Given a system representation $\langle \Pi, C_\Pi, S_\Pi \rangle$, suppose the following properties for this system:

Property 6.1 (**Byzantine Partition**) *For every partition* (A, B, C) *of* Π, *at least one of* A, B, *or* C *contains a core.*

Property 6.2 (**Byzantine Intersection**) $\forall s_i, s_j \in S_\Pi$, $\exists c_k \in C_\Pi$, *such that* $c_k \subseteq (s_i \cap s_j)$.

We want to show that these two properties are equivalent. Before doing so, we prove two preliminary lemmas, which are useful in the proof of the equivalence between properties 6.1 and 6.2. For convenience, we define $f : x \in \Pi \rightarrow \{s_1, s_2, \cdots, s_k\} \subseteq S_\Pi$ as a function that evaluates to the survivor sets x belongs to. Thus, given a subset of processes X, we define S_X as follows:

$$S_X = \cup_{x \in X} f(x) \tag{1}$$

Lemma 6.3 *Let* (A, B, C) *be a partition of* Π *such that none of* A, B, *or* C *contains a core. Suppose that for all* $s \in S_\Pi$, *there is a* $c \in C_\Pi$ *such that* $c \subseteq s$. *Then, we have that for all* $s \in S_{Pi}$, $(s \nsubseteq A) \wedge (s \nsubseteq B) \wedge (s \nsubseteq C)$

Proof: The proof is straightforward. If one of A, B, or C contains a survivor set, then it also contains a core, because all survivor sets contain a core. This contradicts our assumption that none of the partitions contains a core. $\square_{6.3}$

Lemma 6.4 *Let* S_Π *be such that* $\forall s_i \in S_\Pi$, $\exists c_j \in C_\Pi$ *such that* $c_j \subseteq s_i$. *Given a partition* (A, B, C) *of* Π, *such that none of* A, B, *or* C *contain a core, the following properties hold:*

> *6.4.1* $\forall I \in \{A, B, C\}$, $(S_\Pi \nsubseteq S_I)$;
>
> *6.4.2 For all permutations* I, J, K *of* $\{A, B, C\}$, $\exists s_i \in S_\Pi$, *such that* $(s_i \in ((S_I \cap S_J) - S_K))$.

Proof:

- 6.4.1: Suppose we have a subset $\Gamma \subseteq \Pi$ such that for all $s \in S_\Pi$ we have that $R \cap s \neq \emptyset$. By the defined relation between cores and survivor sets, there is a subset of processes $c \in C_\Pi$ such that $c \subseteq \Gamma$. Thus, if $S_\Pi = S_I$, then by our previous observation, I contains a core.
- 6.4.2: we prove this property by contradiction. Suppose without loss of generality that $((S_A \cap S_B) - S_C) = \emptyset$. We prove that for all $s \in S_\Pi$, we have that $s \in S_C$. There are three cases to be considered:
  1. if $s \in (S_A \cap S_B)$, then by assumption it is in S_C;
  2. if $(s \in S_A) \wedge (s \notin S_B)$, then by lemma 6.3 $s \in S_C$;
  3. if $(s \notin S_A) \wedge (s \notin S_B)$, then $s \subseteq C$, which violates lemma 6.3.

If C contains at least one element from every survivor set, then, by property 6.4.1, C contains a core. This contradicts our assumption that none of the partitions contains a core.

$\square_{6.4}$

Claim 6.5 Byzantine Partition ≡ Byzantine Intersection.

Proof:

- Byzantine Partition → Byzantine Intersection

  We prove this implication by contradiction. Assume that property 6.1 holds and there are two survivor sets $s_i, s_j \in S_\Pi$ such that $(s_i \cap s_j)$ does not contain a core. We need to build a partition (A, B, C) such that none of the subsets contain a core. Suppose the following partition: $A = \Pi - s_i$, $B = (s_i \cap s_j)$, and $C = (s_i - B)$. Subset A cannot contain a core, because it has no element from s_i. By assumption, B does not contain a core either. Because C contains no elements from s_j, we have that C also does not contain a core. Thus, none of A, B, or C contain a core, contradicting our assumption that property 6.1 holds.

- Byzantine Intersection → Byzantine Partition

  We prove this implication also by contradiction. Assume that property 6.2 holds and there is a partition (A, B, C) such that neither A, B, nor C contain a core. From lemma 6.4, we have that:

$$\exists x_1 \in S_A, \text{ such that } x_1 \in (S_A \cap S_B) - S_C \tag{2}$$

$$\exists x_2 \in S_A, \text{ such that } x_2 \in (S_A \cap S_C) - S_B \tag{3}$$

  Because $x_1 \notin S_C$ and $x_2 \notin S_B$, we have that $(x_1 \cap x_2) \subseteq A$. By assumption, A does not contain a core, and consequently $x_1 \cap x_2$ does not contain a core. This contradicts, however, our assumption that property 6.2 holds.

$\square_{6.5}$

6.1 Lower bound on process replication

The intersection (partition) property is necessary and sufficient for solving Strong Consensus in a synchronous system with byzantine failures. First, we prove that this property is necessary. The proof we provide is based upon the one by Lamport for independent and identically distributed process failures [25, 26]. We show that if there is a partition of the processes in three non-empty subsets, such that none of them contains a core, then there is at least one run in which agreement is violated, for any algorithm $\mathcal{A}$. This is illustrated in figure 3, where we have three executions: α, β, and γ. Suppose that we have a system representation $\langle \Pi, C_\Pi, S_\Pi \rangle$ and a partition of Π in three non-empty subsets (A, B, C) such that none of them contains a core. In addition, suppose by way of contradiction that we have an algorithm $\mathcal{A}$ that solves Strong Consensus in such a system.

In execution α, the initial value of every the processes is the same, let's say v. Moreover, all the processes in subset B are faulty, and they all lie to the processes in subset C about their initial values and the value received from processes in A. Thus, running algorithm $\mathcal{A}$ in such a execution results in all the processes in subset C deciding v, by the strong validity property. Execution β is analogous to execution A, but instead of every process beginning with a initial value v, they all have initial value $v' \neq v$. Consequently, by the strong validity property, all processes in B decide v' in this execution. Lastly, in execution γ, the processes in subset C have initial value v, whereas processes in subset B have initial value v'. The processes in subset A are all faulty and behave for processes in C as in execution α. For processes in C, however, processes in B behave as in execution β. Because processes in C cannot distinguish executions α from γ, processes in C have to decide v. At the same time, processes in B cannot distinguish executions β from γ, and therefore they decide v'. Consequently, there are correct processes which decide differently in execution γ, violating the agreement property of Strong Consensus.

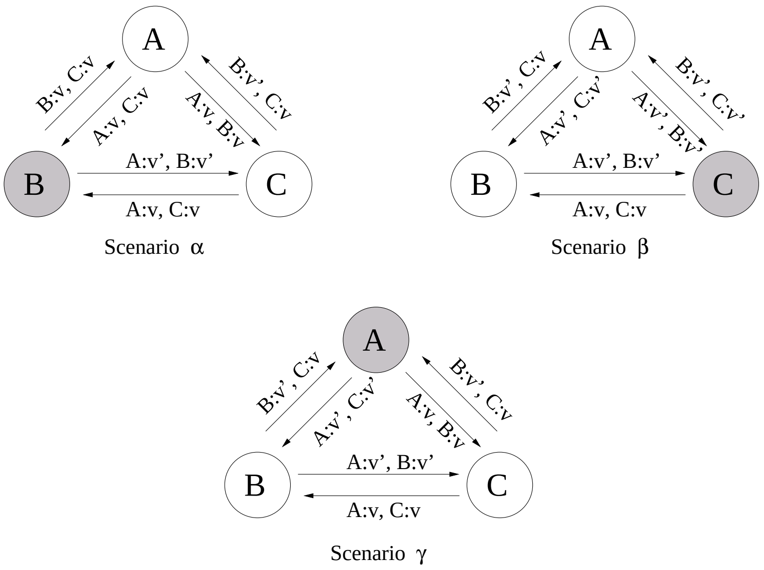

Figure 3: Executions illustrating the violation of Consensus. The processes in shaded subsets are all faulty in the given execution

We now provide a more formal argument by proving the following theorem. Before proceeding in the statement and proof of the theorem, we introduce some useful notation. Let α be an execution. We assume that $\alpha(i_0 i_2 \cdots i_k)$ is the value that process p_{i_0} receives from process p_{i_1}, which claims that this value is the initial value of p_k passed by every process p_i to process p_{i-1} in this k-process chain. For example, $\alpha(ijk)$ is the value that process p_i receives from process p_j, which is the value that supposedly p_k has sent to p_j as its initial value. If the k-process chain contains only correct process, $k \geq 1$, then the value $\alpha(i_0 i_2 \cdots i_k)$ is the initial value of p_k. Otherwise, this property is not guaranteed. In the case that $k = 1$, we have that $\alpha(i)$ is the initial value of process p_i.

Theorem 6.6 *Let sys* $= \langle \Pi, C_\Pi, S_\Pi \rangle$ *be a system representation. If there is a partition* (A, B, C) *of* Π *such that none of* A, B, *or* C *contains a core, then there is no algorithm which solves Strong Consensus in such a system.*

Proof: We assume without loss of generality that none of A, B, or C is empty.

Suppose there is an algorithm $\mathcal{A}$ which solves Strong Consensus in *sys*. We construct recursively an execution in which two correct processes decide differently. Moreover, the agreement violation in this execution is independent of the number of rounds the algorithm runs. Even if the algorithm runs for an infinite number of rounds, it cannot prevent agreement violation.

By assumption, there is a partition (A, B, C) of Π in three non-empty subsets such that none of A, B, or C contains a core. Let's start by describing two preliminary executions that are used to construct the one in which agreement is violated. We construct executions α and β as follows:

$$\begin{aligned}
&\text{Let } a \in A,\ b \in B,\ c \in C,\ v \in V,\ v' \in V,\ v' \neq v\\
&\qquad \alpha(a) = \alpha(b) = \alpha(c) = v\\
&\qquad \beta(a) = \beta(b) = \beta(c) = v'
\end{aligned}$$

$$\begin{aligned}
&\text{Let } w \in \Pi^* \text{ and } p \in \Pi\\
&\qquad \alpha(paw) = \alpha(aw)\\
&\qquad \alpha(abw) = \alpha(bw)\\
&\qquad \alpha(cbw) = \beta(bw)\\
&\qquad \alpha(pcw) = \alpha(cw)\\
&\qquad \beta(paw) = \beta(aw)\\
&\qquad \beta(pbw) = \beta(bw)\\
&\qquad \beta(acw) = \beta(cw)\\
&\qquad \beta(bcw) = \alpha(cw)
\end{aligned}$$

Based on executions α and β, we constructed execution γ as follows:

$$\begin{aligned}
&\text{Let } a,\ b,\ c,\ v,\ v',\ p, \text{ and } w \text{ be as in definition of executions } \alpha \text{ and } \beta\\
&\qquad \gamma(a) = v\\
&\qquad \gamma(b) = v'\\
&\qquad \gamma(c) = v\\
&\\
&\qquad \gamma(baw) = \beta(aw)\\
&\qquad \gamma(caw) = \alpha(aw)\\
&\qquad \gamma(pbw) = \gamma(bw)\\
&\qquad \gamma(pcw) = \gamma(cw)
\end{aligned}$$

It remains to show that $\alpha(cw) = \gamma(cw)$ and $\beta(bw) = \gamma(bw)$, for $b \in B$, $c \in C$, and $w \in \Pi^*$. We prove these equivalences by a simple induction on the length of w.

- Base case: $|w| = 0$
  For $|w| = 0$, we have that $\alpha(c) = v = \gamma(c)$ and that $\beta(b) = v' = \gamma(b)$.
- Induction step: the induction hypothesis is that the proposition is valid for all w such that $|w| \leq i$. We need to prove that the proposition is true for all w of length

$i+1$. That is, we need to show that $\alpha(cpw) = \gamma(cpw)$ and $\beta(bpw) = \gamma(bpw)$ for every $p \in \Pi$. There are three cases to be analyzed: $p = a$, $p = b$, and $p = c$. We show below these three cases separately:

1. $p = a$: by the definitions of α, β, and γ:

$$\alpha(caw) = \alpha(aw) = \gamma(caw)$$
$$\beta(baw) = \beta(aw) = \gamma(baw)$$

2. $p = b$: by the definitions of α, β, and γ and the induction hypothesis:

$$\alpha(cbw) = \beta(bw) = \gamma(bw) = \gamma(cbw)$$
$$\beta(bbw) = \beta(bw) = \gamma(bw) = \gamma(bbw)$$

3. $p = c$: by the definitions of α, β, and γ and the induction hypothesis:

$$\alpha(ccw) = \alpha(cw) = \gamma(cw) = \gamma(ccw)$$
$$\beta(bcw) = \alpha(cw) = \gamma(cw) = \gamma(bcw)$$

Because processes in C cannot distinguish between executions α and γ, these processes have to decide v in γ. On the other hand, processes in B cannot distinguish execution β from execution γ, and consequently they have to decide v' in γ. By assumption, in execution γ, the processes in both subset B and subset C are correct. Therefore, the agreement property of Strong Consensus is violated in this execution.

$\Box_{6.6}$

6.2 An algorithm to solve Strong Consensus

We describe an algorithm that solves Strong Consensus in a system $sys = \langle \Pi, C_\Pi, S_\Pi \rangle$ which satisfies the intersection property. This algorithm is based on the one described by Lamport to demonstrate that it is sufficient to have $3t+1$ processes (t is the maximum tolerated number of faulty processes) to have interactive consistency in a setting with byzantine processes [25].

In our algorithm, all the processes run the same state machine. Every process creates a tree where every node is labeled with a string w of process id's and stores a value. Every label is composed of a sequence of process id's and each id appears at most once in a given label w. The value stored at a given node labeled w corresponds to the value forwarded by the chain of processes with id's on the string, following the sequence determined by the string. Thus, at round r, every correct process p_i sends a message containing the values stored at depth r of the tree to all the other processes. Every correct process p_j that receives this message at round $r+1$ stores the values contained in it in the following manner: for every node labeled wi, with $w \in Pid^*, |w| = r$, make the value of node equal to the value in the message sent by p_i corresponding to w.

A simple example will help to clarify the use of the tree. Suppose that a correct process p receives at round 3 a message from process p_k, which contains the string ij and the value v associated to this string. Process p hence stores the value v at the node labeled ijk and forward a message containing ijk associated to the value v to all the other processes.

An important observation about the tree built by the algorithm is that the last level is composed of survivor sets. More specifically, a $Node(w)$[4] is a leaf if and only $\Pi - Processes(w)$ does not contain a survivor set [5]. Consequently, if $Node(wp)$ is a leaf, then $Child(w)$[6] is a survivor set [7]. A property that every node of the tree labeled w satisfies is that $\Pi - Processes(w)$ has to contain a survivor set. A consequence of the previous observations is that the depth of tree is $|\Pi| - \min|s_i||s_i \in S_\Pi + 1$. An example of a tree is presented in figure 4, for a system a characterized by the following sets:

- $\Pi = \{a, b, c, d, e\}$
- $C_\Pi = \{ab, ac, ad, ae, bc, bd, cd, ce, de\}$
- $S_\Pi = \{abce, abde, acd, bcde\}$

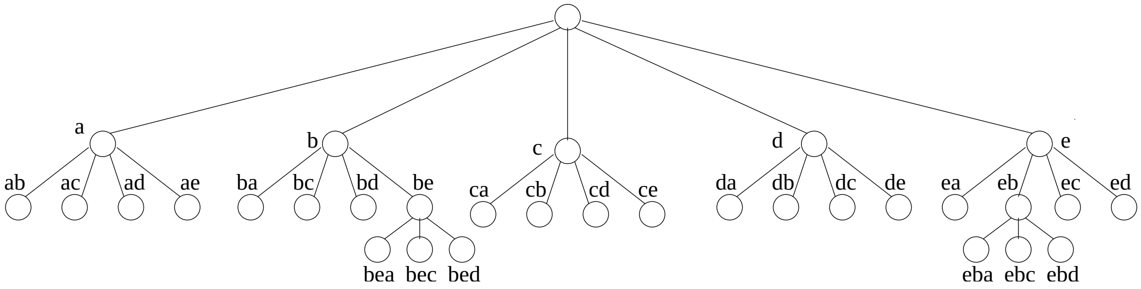

Figure 4: An example of a tree built by each process in the first stage of the algorithm.

Building and initializing the tree corresponds to the first stage of the algorithm. The second stage consists in running several rounds of message exchange. In the first round, each process broadcast its initial value. In the subsequent rounds, each process broadcast the values it learned in the previous round. As the processes receive the messages containing values learned in previous rounds, each node fills out the nodes of its tree with these values. Because the depth of the tree is $|\Pi| - \min|s_i||s_i \in S_\Pi + 1$, this is exactly the total number of rounds required for message exchanging. An important observation is that this matches the lower bound on the number of rounds necessary to solve Consensus in a byzantine setting. As shown in [9], if t is the maximum number of process failures assumed, $t \leq (|\Pi| - 2)$, then at least $t + 1$ rounds are necessary. Furthermore, the proof presented does not assume independent and identically distributed process failures, and therefore it accommodates a more general model as ours. A question that may strike one's mind is why we cannot use a trick of using a subset of cores or survivor sets to design

---

[4] $Node(w)$ is defined as the node of the tree labeled with the string w.

[5] $Processes(w)$= $\{p|p.id$ is in $w\}$

[6] $Child(w)$= $\{p_i|$node labeled wi is a child of node labeled $w\}$.

[7] Observe that the tree structure is the same for all correct processes, and hence none of *Processes*$(\cdot)$,*Node*$(\cdot)$, or *Child*$(\cdot)$ need to be associated with any particular process.

an algorithm that runs in fewer rounds, as we did for the synchronous crash model. The answer is simple: from our previous results on the lower bound for process replication, this subset would need to satisfy the byzantine intersection property. If we take a core as an isolated system, for instance, then it clearly does not satisfy this property.

Finally, in the last stage, each process traverses the tree visiting the nodes in postorder to decide on a value. We show later in this section that all processes decide on the same value after traversing the tree.

Before presenting the pseudo-code of the algorithm, a few words about the notation. We define *Pid* to be the set of process id's, i.e., $Pid = \{i | (i = \text{p.id}) \wedge (p \in \Pi)\}$. This is convenient, because we label the nodes of the trees with strings of process id's. The function $x.\mathit{Value}(w)$ evaluates to the value v associated to the string of id's w. Because v is provided either by a message or a node of the tree, the value x represents either a process or a message. Thus, $m.\mathit{Value}(w)$ evaluates to the value v that message m carries associated to string w, whereas $p_i.\mathit{Value}(w)$ evaluates to the value v stored by node labeled w at process p_i. This is a slight abuse of notation, but it is convenient and the differentiation between the cases will be clear from context.

A pseudo-code of the algorithm is presented below.

We now prove that the algorithm **SyncByz** satisfy the properties of Strong Consensus. First, we state and prove three preliminary lemmas that we are useful in demonstrating that these properties hold for **SyncByz**.

For the following lemmas, suppose that S_{min} is a minimum-sized survivor set in S_Π. That is, there is no survivor set in S_Π with fewer elements than S_{min}.

Lemma 6.7 *Let α be an execution of* **SyncByz**, *p_i be a correct process in α, and $w \in \text{Pid}^*$ be the label of some non-leaf node. At the end of round $r = (|\Pi| - |S_{\text{min}}| + 1)$, for every $p_k, p_j \in \text{Correct}(\alpha)$, $p_j.\text{Value}(wi) = p_k.\text{Value}(wi) = v_i^w$, where $v_i^w \in V$ is the value $p_i.\text{Value}(w)$ at round $|w|$.*

Proof: Let $s_c \in S_\Pi$ be a survivor set containing only correct processes in α.

We prove this lemma by recursion on the length of node label w, $1 \leq |w| \leq (|\Pi| - |S_{min}| + 1)$. For the base case, suppose that wi is the label of a leaf. If p_i is correct, then it forwards the same value $v_i^w \in V$ it has for w to all the other processes at round $|w| + 1$. Notice that if $w = \emptyset$, then p_i sends its initial value. Thus, for every process $p_j \in \mathit{Correct}(\alpha)$, $p_j.\mathit{Value}(wi) = v_i^w$ at the end of round $r = |w| + 1$, where $v \in V$ is the value $p_i.\mathit{Value}(w)$ at round $|w| + 1$.

We now assume that for every $p_i, p_j \in \mathit{Correct}(\alpha)$, $p_j.\mathit{Value}(wi) = v_i^w$, $|wi| \leq |w'| \leq (|\Pi| - |S_{min}| + 1)$, where $v_i^w \in V$ is the value $p_i.\mathit{Value}(w)$ at round $|w| + 1$. We need to prove the proposition for the labels of length $|w|$. Suppose that $w = w'i$. Let s_1 be such that $s_1 \subseteq \mathit{Child}(w)$. From the inductive assumption, for every process $p_{i_1} \in s_c \cap s_1$ and $p_j \in \mathit{Correct}(\alpha)$, we have that $p_j.\mathit{Value}(wi_1) = v_i^{w'}$, where $v_i^{w'} \in V$ is the value $p_i.\mathit{Value}(w')$ at round $|w'| + 1$. Moreover, suppose that there are two survivor sets $s_2, s_3 \in S_\Pi$, $(s_2 \cap s_3) \neq (s_1 \cap s_c)$, such that $(s_2 \cap s_3) \in \mathit{Child}(w)$. From the byzantine intersection property, there is ate least one correct process $p_{i_3} \in (s_2 \cap s_3)$. Consequently, if for every process $p_{i_4} \in s_c \cap s_d$, $p_j.\mathit{Value}(wi_4) = v'$, then v' has to be equal to v_i^w. Otherwise, the value $p_j.\mathit{Value}(wi_3) \neq v_k$, contradicting the inductive assumption.

According to the algorithm, we have that for every $p_j \in \mathit{Correct}(\alpha)$, $p_j.\mathit{Value}(w'i) = v_i^{w'}$, where $v_i^{w'} \in V$ is the value $p_i.\mathit{Value}(w')$ at round $|w'| + 1$. $\Box_{6.7}$

Before stating and proving the following lemma, we need to introduce some more notation. We define $\mathit{RLeaves}(w)$ as the set of labels ww', such that $\mathit{Child}(ww') = \emptyset$ and $w' \in \mathit{Pid}^*$.

```
Algorithm SyncByz for process $p_i$:

Input: a set of processes $\Pi$; a set of cores $C_\Pi$; a set of survivor sets $S_\Pi$; an input value $v_i \in V$

Variables:
Let $s_{min}$ be a smallest survivor set in $\mathcal{S}$
Let $r$ be the current round number
Let $root$ be a reference to the root of process $i$'s tree
Let $M$ be a set of messages
Let $P, P'$ be sets of pairs $\langle w, v\rangle$, where $w \in Pid^*$, and $v \in V$

initialization:
    $root \leftarrow$ CreateNode($\emptyset$, $v_i$)
    BuildTree($root$)
    $P \leftarrow \{\langle \emptyset, v_i\rangle\}$

rounds $1 \leq r < (|\Pi| - |s_{min}| + 1)$:
    $SendAll(i, P)$
    let $M$ be the set of messages received by $p_i$ at round $r$
    $P \leftarrow \emptyset$
    for every message $m = (j, P') \in M$ do
        for every node at depth $r$ labeled $wj$, $w \in Pid^*$, $|w| = r$ do
            $p_i.Value(wj) \leftarrow m.Value(w)$
            if node labeled $wj$ is not a leaf then $P \leftarrow P \cup \{\langle wj, m.Value(w)\rangle\}$

round $r = (|\Pi| - |s_{min}| + 1)$:
    $SendAll(i, P)$
    let $M$ be the set of messages received by $p_i$ at round $r$
    for every message $m = (j, P') \in M$ do
        for every node at level $r$ labeled $wj$, $w \in Pid^*$, $|w| = r$, do
            $p_i.Value(wj) \leftarrow m.Value(w)$

    Traverse Tree in postorder, executing the following steps when visiting a node labeled $w$:
        if $Child(w) \neq \emptyset$
        then let $I \leftarrow Child(w)$
            if($\exists s_1, s_2 \in S$ such that $((s_1 \cap s_2) \subseteq I) \wedge (\forall p_j \in (s_1 \cap s_2), p_i.Value(wj) = v, v \in V))$)
            then $p_i.Value(w) \leftarrow v$
            else $p_i.Value(w) \leftarrow \perp$

Auxiliary function
Function BuildTree($w$)
    let $\Gamma \leftarrow Processes(w)$
    $\forall p_j \in \Pi$ such that $p_j \notin \Gamma$
            if ($\exists s_1 \in S$ such that $s_1 \subseteq (\Pi - \Gamma)$)
            then $node \leftarrow$ CreateNode($wj$, $\perp$)
                $Child(w) \leftarrow Child(w) \cup \{node\}$
                BuildTree($wj$)
```

Figure 5: Synchronous Consensus for Dependent Arbitrary Failures

Lemma 6.8 *Let α be an execution of* **SyncByz**, *and u be a node labeled wi, $w \in \text{Pid}^*, p_i \in \Pi$. If for every $wiw' \in \text{RLeaves}(wi)$, it is the case that $\text{Correct}(\alpha) \cap \text{Processes}(iw') \neq \emptyset$, then $p_j.\text{Value}(wi) = p_k.\text{Value}(wi)$ for all $p_j, p_k \in \text{Correct}(\alpha)$ at the end of round $r =$*

$(|\Pi| - |s_{\min}| + 1)$.

Proof: We prove this lemma by induction on the height of the tree, starting from the leaves.

The base case occurs when u is a leaf. By assumption, p_i is correct. Thus, we have that $p_k.\mathit{Value}(wi) = p_l.\mathit{Value}(wi)$, from lemma 6.7.

The induction hypothesis is that the proposition is valid for all the nodes at depth d, starting from the leaves. We need to prove the proposition for a node v at depth $d-1$. We have two cases to analyze: p_i is correct and p_i is faulty. If p_i is correct, then the proof is straightforward from lemma 6.7. We need to analyze the case in which p_i is faulty.

Suppose that p_i is faulty and that every leaf labeled wiw' is such that $\mathit{Processes}(iw') \cap \mathit{Correct}(\alpha) \neq \emptyset$. In this case, for every child labeled wii_1, we have that for all $wii_1w'' \in \mathit{RLeaves}(wii_1)$, $\mathit{Processes}(i_1w'') \cap \mathit{Correct}(\alpha) \neq \emptyset$. By the induction hypothesis, it is the case that $p_j.\mathit{Value}(wii_1) = p_k.\mathit{Value}(wii_1)$ for every $p_{i_1} \in \mathit{Child}(wi)$. From the algorithm, it has to be the case that $p_k.\mathit{Value}(w) = p_l.\mathit{Value}(w)$, for all $p_j, p_k \in \mathit{Correct}(\alpha)$. $\Box_{6.8}$

Lemma 6.9 *Let α be an execution of* **SyncByz**. **SyncByz** *satisfies Strong Validity in α.*

Proof: By the definition of S_Π, in every execution there is at least one survivor s_i set containing only correct processes. From lemma 6.7, for every process $p_i \in s_a$, we have that $p_j.\mathit{Value}(i)$ is the initial value of p_i, assuming p_j is correct. If all the processes start an execution with the same initial value v, then, from the algorithm and the assumption that the intersection property holds, $p_j.\mathit{Value}(\emptyset)=v$. $\Box_{6.9}$

Lemma 6.10 *Let α be an execution of* **SyncByz**. **SyncByz** *satisfies Agreement in α.*

Proof: Let p_i be a process in Π, and α be some execution of **SyncByz**. We need to prove that for every process $p_j \in \mathit{Correct}(\alpha)$, $p_j.\mathit{Value}(\emptyset) = v$, for some decision value $v \in V \cup \{\bot\}$. By the construction of the tree, for every leaf labeled iwj, $w \in (\mathit{Pid} - \{i,j\})^*$, there is at least one correct process $p_{i_1} \in \mathit{Processes}(iwj)$. From lemma 6.2, we have that by the end of round $r = (|\Pi| - |x| + 1)$, for some $v \in V \cup \{\bot\}$, $p_{i_1}.\mathit{Value}(i) = v$, for all $p_{i_1} \in \mathit{Correct}(\alpha)$.

From the previous argument, we have that for every $p_{i_2}, p_{i_3} \in \mathit{Correct}(\alpha)$ and every $p_{i_4} \in \Pi$, $p_{i_2}.\mathit{Value}(i_4) = p_{i_3}.\mathit{Value}(i_4)$. According to the algorithm, the decision value of every correct process therefore has to be the same. This proves that the agreement property holds for **SyncByz**. $\Box_{6.10}$

Lemma 6.11 *Let α be an execution of* **SyncByz**. **SyncByz** *satisfies Termination in α.*

Proof: The absence of infinite loops in the algorithm makes it straightforward to observe that it eventually terminates and every process eventually decides. $\Box_{6.11}$

7 Asynchronous Consensus with Arbitrary Failures

Under Construction

8 Final Remarks

The results we showed in this paper encourage one to use cores and survivor sets in the design of fault-tolerant algorithms. There are a few questions, however, that remain to be answered. First, it is not clear that cores or survivor sets are a good way of modeling failure correlation. In the worst case, there is an exponential number of such subsets. Representing and finding cores or survivor sets in these system configurations may not be practical. Some of our results show that even in the case that there is an exponential number of cores in a system, just a subset of cores are necessary to satisfy replication requirements. For example, in the case of Consensus for synchronous systems with crash failures, processes need to know about a single core. For asynchronous systems with crash failures, all is needed is a set of survivor sets that is a coterie. Both cases imply that not all subsets are needed, but just some of them.

A second question is how to extract the information about cores. One has to know how to correlate failures in order to determine cores. An obvious approach is to consider failure probabilities. This may not be as practical as assuming independent failure probabilities, because in general one has to deal with equations with an exponential number of terms. Alternatively, one can use intrinsic properties of the system to correlate process failures. For example, if there are two PC's in the same room, then a power failure can make both crash at the same time. Another example is having implementations using the same buggy code. Processes running such a software may present the same arbitrary behavior and consequently present correlated failures. Thus, it is not necessary to quantify failure correlation in order to determine cores in a system. Although we do not have a nice and closed formula to compute cores in the general case, there are heuristics that can be used on a per-case basis. We present two heuristics in [27].

In more dynamic systems, there is the issue of correlating failures on-line. Suppose the case of mobile nodes. Assuming each mobile node is a process, processes close to each other may be subject to the same unfortunate events. In this case, it is necessary to know the position of the nodes to determine cores. Furthermore, cores are constantly changing. Thus, a probing mechanism is necessary to determine positioning information. This information is then used to extract cores. A probing mechanism, however, is not sufficient. It is also necessary to have either an agreement protocol so that processes agree on the cores at a given point of an execution, or protocols should be designed to cope with inconsistencies in the set of cores across all processes.

Generalizing the results we have is also one of our goals. It seems that the idea of using protocols proposed in the literature modified to consider cores or survivor sets is not applicable only to Consensus. So far we have investigated the application of our model only to Distributed Consensus yet we plan to do the same for other problems in FT distributed computing. By doing this, we will gain more intuition on the applicability of our model.

To conclude, we believe that all questions we posed here are important and that we will have answers for most of them only after applying to the designing of real systems. We are optimistic about our results, because the ones we have so far show several benefits in using failure correlation in the design of algorithms and the preliminary results we have about cores in real systems show that tha approach is not unrealistic.

References

[1] I. Keidar and S. Rajsbaum, "On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial," Tech. Rep. MIT-LCS-TR-821, MIT, May 2001.

[2] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, pp. 225–267, March 1996.

[3] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies*, pp. 43–98, Princeton University Press, 1956.

[4] J. Wensley, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," in *Proceedings of the IEEE*, vol. 66, pp. 1240–1255, October 1978.

[5] R. Rodrigues, B. Liskov, and M. Castro, "BASE: Using Abstraction to Improve Fault Tolerance," in *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, vol. 35, (Chateau Lake Louise, Banff, Alberta, Canada), pp. 15–28, October 2001.

[6] Y. Ren and J. B. Dugan, "Optimal Design of Reliable Systems Using Static and Dynamic Fault Trees," *IEEE Transactions on Reliability*, vol. 47, pp. 234–244, December 1998.

[7] P. Thambidurai and Y.-K. Park, "Interactive Consistency with Multiple Failure Modes," in *IEEE 7th Symposium on Reliable Distributed Systems*, (Columbus, Ohio), pp. 93–100, October 1988.

[8] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," in *29th ACM Symposium on Theory of Computing*, pp. 569–578, may 1997.

[9] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, 1998.

[10] K. Kihlstrom, L. Moser, and P. M. Melliar-Smith, "Solving Consensus in a Byzantine Environment using an Unreliable Failure Detector," in *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'97)*, (Chantilly, France), pp. 61–76, December 1997.

[11] D. Malkhi and M. Reiter, "Unreliable Intrusion Detection in Distributed Computations," in *Proceedings of the 10th Computer Security Foundations Workshop (CSFW97)* , (Rockport, MA), pp. 116–124, June 1997.

[12] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *Journal of the ACM*, vol. 1, pp. 77–97, January 1987.

[13] S. Mullender, ed., *Distributed Systems*, ch. 5. Addison-Wesley, 2nd ed., 1995.

[14] B. Charron-Bost, R. Guerraoui, and A. Schiper, "Synchronous System and Perfect Failure Detector: solvability and efficiency issues," in *IEEE International Conference on Dependable Systems and Networks (DSN'00)*, (New York, NY), pp. 523–532, June 2000.

[15] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, pp. 374–382, April 1985.

[16] T. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *Journal of the ACM*, vol. 43, pp. 685–722, July 1996.

[17] A. Doudou and A. Schiper, "Muteness Detectors for Consensus with Byzantine Processes," in *Proceedings of the 17th ACM Symposium on Principle of Distributed Computing*, (Puerto Vallarta, Mexico), p. 315, July 1998. (Brief Announcement).

[18] B. Charron-Bost and A. Schiper, "Uniform Consensus is Harder Than Consensus," tech. rep., École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[19] D. Skeen, "Determining the Last Process to Fail," *ACM Transactions on Computer Systems*, vol. 3, pp. 15–30, February 1985.

[20] L. Lamport and M. Fischer, "Byzantine Generals and Transaction Commit Protocols," tech. rep., SRI International, April 1982.

[21] D. Dolev, R. Reischuk, and H. R. Strong, "Early Stopping in Byzantine Agreement," *Journal of the ACM*, vol. 37, pp. 720–741, October 1990.

[22] F. Junqueira and K. Marzullo, "Lower Bound on the Number of Rounds for Synchronous Consensus with Dependent Process Failures," tech. rep., UCSD, La Jolla, CA, September 2002. http://www.cs.ucsd.edu/users/flavio/Docs/lb.ps.

[23] R. Guerraoui and A. Schiper, "Consensus Service: A Modular Approach for Building Fault-tolerant Agreement Protocols in Distributed Systems," in *26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, (Sendai, Japan), pp. 168–177, June 1996.

[24] A. Schiper, "Early Consensus in a Asynchronous System with a Weak Failure Detector," *Distributed Computing*, vol. 10, pp. 149–157, April 1997.

[25] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[26] M. Pease, R. Shostak, , and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, vol. 27, pp. pp. 228–234, April 1980.

[27] F. Junqueira, K. Marzullo, and G. Voelker, "Coping with Dependent Process Failures," tech. rep., UCSD, La Jolla, CA, December 2001. http://www.cs.ucsd.edu/users/flavio/ Docs/JuMaVo2001.ps.