Synchronous 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 92093-0114, USA

Abstract

We present a new abstraction to replace the t of n assumption used in designing fault-tolerant algorithms. This abstraction models dependent process failures yet it is as simple to use as the t of n assumption. To illustrate this abstraction, we consider Consensus for synchronous systems with both crash and arbitrary process failures. By considering failure correlations, we are able to reduce latency and enable the solution of Consensus for system configurations in which it is not possible when forced to use algorithms designed under the t of n assumption. We show that, in general, the number of rounds required in the worst case when assuming crash failures is different from the number of rounds required when assuming arbitrary failures. This is in contrast with the traditional result under the t of n assumption.

Keywords: Distributed algorithms, Fault tolerance, Correlated failures, Consensus, Synchronous systems

1 Introduction

Most fault-tolerant algorithms 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 $\diamond W$ [2, 11].

The use of the t of n assumption dates back to the earliest work on fault-tolerant computing [22]. It was first applied to distributed coordination protocols in the SIFT project [23] 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 [19]. 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 algorithm designers. If one wishes to apply, for example, a Consensus algorithm in some real distributed system, one can use one of two approaches:

1. Use some off-line analysis technique, such as fault tree analysis [18] to identify how processes fail in a corre-

*This work was developed in the context of RAMP, which is DARPA project number N66001-01-1-8933.

†The author is currently being funded by CAPES (process number BEX1346/98-9) and AFOSR MURI grant 411316865.

lated 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 [18]. 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 algorithm.

We believe that our approach and algorithms 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 an algorithm. 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 problem under the synchronous system assumption and for both crash and arbitrary failures. We derive a new lower bound for the number of rounds to solve Consensus and show that it takes, in general, fewer rounds for crash failures in the worst case than for arbitrary failures. This is in contrast to the t of n assumption, where the number of rounds required in the worst case is the same for both failure models [11, 13]. We show that these bounds are tight by giving algorithms that meet them. These algorithms are fairly simple generalizations of well-known algorithms, which is convenient. Proving them correct gave us new insight into the structure of the original algorithms. 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, [21]) 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 algorithms 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* [14]. This abstraction allows one to define quorums that take correlated failures into account, and it has been used to express a sufficiency condition for replication. Our work can be seen as generalizing this work that has been done for quorum systems in that we give replication requirements for dependent failures for another set of problems, namely Consensus. Hirt and Maurer generalize the t of n assumption in the context of secure multi-party computation [7]. With *collusion* and *adversary* structures, they characterize subsets of players that may collaborate to either disrupt the execution of an algorithm or obtain private data of other players. Because our focus is not on malicious behavior, the abstractions we propose are defined differently and fulfill a distinct set of properties.

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. In Section 4, we state a theorem that generalizes the lower bound on the number of rounds in our model. Sections 5 and 6 describe tight weakest replication requirements and Consensus algorithms for crash and arbitrary failures, respectively. A discussion on the implementation of our abstraction in real systems is provided in Section 7. Finally, we draw conclusions and discuss future work in Section 8.

Due to lack of space, most proofs and several details of the system model are omitted. For a more detailed discussion on this work, we refer the interested reader to [8, 9, 10].

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 the 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., $\mathit{Pid} = \{i : p_i \in \Pi\}$. We use this set to define a sequence w of process id's. Such a sequence w is an element of Pid^*.

A process communicates with others by exchanging messages. Messages are transmitted through point-to-point reliable channels[1], and each process is connected to every other process through one of these channels. Processes, on the other hand, are not assumed to be reliable. We consider both crash and arbitrary process failures. In contrast to most previous works in fault-tolerant distributed systems, process failures are allowed to be correlated.

Each process $p \in \Pi$ executes a deterministic automaton as part of the distributed computation [1, 2]. A deterministic automaton is composed of a set of states, an initial state,

[1] In this paper, a reliable channel is one that satisfies the following properties: 1) Every message sent from a correct process p_i to a correct process p_j is eventually received by p_j, and exactly once; 2) A correct process p_j receives a message m from a process p_i only if p_i has sent it to p_j.

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 atomic, and steps of different processes are allowed to overlap in time. Additionally, we assume that executions can be split into synchronous rounds. In each round, a process: 1) Sends messages at the beginning of the round; 2) Receives messages that other processes send at the beginning of the round; 3) Changes its state. The algorithms we describe in this paper hence proceed in rounds.

2.1 Replacing the t of n Assumption: Cores and Survivor Sets

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.

Assuming that failed processes do not recover, to achieve fault-tolerance in a system with a set of processes Π, it is necessary to guarantee in every execution that a non-empty subset of Π survives. A process is said to survive an execution if and only if it is correct throughout that execution. Thus, we would like to distinguish subsets of processes such that the probability of all processes in such a subset failing is negligible. Moreover, we want these subsets to be minimal in that removing any process of such a subset c makes the probability that the remaining processes in c fail not negligible. We call these minimal subsets *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's characterization. We present in Section 7 a discussion on the problem of finding cores.

By assumption, at least one process in each core will be correct in an execution. Thus, a subset of processes that has a non-empty intersection with every core 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.

Cores and survivor sets generalize subsets under the t of n assumption of size $t+1$ and $n-t$, respectively. Thus, we conjecture that fault-tolerant algorithms that have subsets of size $t+1$ and $n-t$ embedded in its characterization are translatable to our model by simply exchanging these subsets with cores and/or survivor sets. Intuitively, performance may be impacted by such a change, but correctness is preserved. It is also important to observe that both cores and survivor sets describe the worst-case failure scenarios of a system. They are therefore equivalent to *fail-prone systems* as defined by Malkhi and Reiter in their work on Byzantine quorum systems [14]. For a given system configuration, one can obtain the fail-prone system by taking the complement of every survivor set.

We now define cores and survivor sets more formally. Let R be a rational number expressing a desired reliability, 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 rational target degree of reliability* $R \in [0,1]$*, the set of processes* c *is a* core *of* Π *if and only if:*

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

Given a set of processes Π *and a set of cores* C_Π*,* s *is a survivor set if and only if:*

1. $s \subseteq \Pi$*;*
2. $\forall c \in C_\Pi$*:* $s \cap c \neq \emptyset$*;*
3. $\forall p_i \in s, \exists c \in C_\Pi$*:* $((p_i \in c) \wedge ((s - \{p_i\}) \cap c = \emptyset)$*.*

We define C_Π *and* S_Π *to be the set of cores and survivor sets of* Π*, respectively.* $\square_{2.1}$

The function $r(\cdot)$ 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 a probability. If this information is known by other means, then cores can be directly determined. For example, consider a six-computer system, each computer representing a process. Two of them are robust computers (ph_1 and ph_2), located in different sites and managed by different people. The other four computers (pl_1, pl_2, pl_3, pl_4) are located in the same room and are managed by the same person. If machines only fail by crashing due to power outages and software bugs, then we can safely assume that ph_1 and ph_2 are reliable processes and fail independently, whereas pl_1, pl_2, pl_3, and pl_4 are less reliable and their failures are highly correlated. To simplify the analysis, we assume that the failure of process pl_i implies the failure of pl_j, $i \neq j$. In order to maximize reliability, a core is hence composed of three processes: ph_1, ph_2, and pl_i, $1 \leq i \leq 4$. Note that adding another process pl_j, $i \neq j$, does not increase the reliability of a core, and the subset is not minimal in this case. Under these assumptions, we construct the set of cores and survivor sets for this system as follows:

Example 2.2 :

- $\Pi = \{ph_1, ph_2, pl_1, pl_2, pl_3, pl_4\}$
- $C_\Pi = \{\{ph_1, ph_2, pl_1\}, \{ph_1, ph_2, pl_2\}, \{ph_1, ph_2, pl_3\}, \{ph_1, ph_2, pl_4\}\}$
- $S_\Pi = \{\{ph_1\}, \{ph_2\}, \{pl_1, pl_2, pl_3, pl_4\}\}$

$\square_{2.2}$

In the following sections, we assume that, for a given system, these subsets are provided as part of the system's characterization. A system is henceforth described by a triple $\langle \Pi, C_\Pi, S_\Pi \rangle$, where Π is a set of processes, C_Π is a set of cores of Π, and S_Π is a set of survivor sets of Π. All the other assumptions for the system model discussed previously are implicit. From this point on, we call $\langle \Pi, C_\Pi, S_\Pi \rangle$ a *system configuration*, or simply a *system*.

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 decide on the same value. The set of possible decision values is denominated V throughout this paper. For many applications, a binary set V is sufficient, but we assume a set V of arbitrary size, to keep the definition as general as possible.

In the crash failure model, Consensus is often specified in terms of the following three properties [5]:

Validity : If some non-faulty process $p \in \Pi$ decides on value v, then v was proposed by some process $q \in \Pi$;

Agreement : If two non-faulty processes $p, q \in \Pi$ decide on values v_p and v_q respectively, then $v_p = v_q$;

Termination : Every correct process eventually decides.

Validity, as specified above, assumes that no process will ever try to "cheat" on its proposed value. This is true for crash failures, but unrealistic for arbitrary failures. Although a faulty process might not be able to prevent agreement by cheating on its proposed value, it may prevent progress of the system as whole. Thus, for arbitrary failures, *Strong Validity* is often used instead of Validity [15]. 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.

We refer to Consensus with the Strong Validity property as *Strong Consensus*.

4 Lower Bound on the Number of Rounds

Consider a synchronous system in which the t of n assumption holds for process failures. In such a system, t is the maximum number of process failures among all possible executions and f is the number of failures of a particular execution. It is well known that for every synchronous Consensus algorithm $\mathcal{A}$, there is some execution in which some correct process does not decide earlier than $f + 1$ rounds, where $f \leq t \leq n-2$ [1, 3, 12]. Furthermore, there is some execution in which some correct process does not stop earlier than $\min(t + 1, f + 2)$ rounds, for $t \leq n - 2$ [4].

These lower bounds were originally proved for crash failures, but they have to hold for arbitrary failures as well because the model of arbitrary failures is strictly weaker than the model of crash failures. In our model of dependent failures, however, the required number of rounds in general differs between these two failure models.

Before generalizing the lower bound on the number of rounds, we define the term *subsystem*. Let Λ be some predicate expressing the process replication requirement for a given failure model. For example, assuming t of n arbitrary process failures, the replication requirement is $n > 3t$. Thus, we have in this example that $\Lambda =$ "$n > 3t$". Examples of such predicates in our model of dependent failures are provided in Sections 5 and 6. A subsystem of a system that satisfies Λ is then defined as follows:

Definition 4.1 *Let Λ be a replication requirement and* sys $= \langle \Pi, C_\Pi, S_\Pi \rangle$ *be a system configuration satisfying Λ. A system* sys$' = \langle \Pi', C'_\Pi, S'_\Pi \rangle$ *is a subsystem of* sys *if and only if $\Pi' \subseteq \Pi$, $C'_\Pi \subseteq C_\Pi$, and* sys$'$ *satisfies Λ.* $\square_{4.1}$

A subsystem *sys*$'$ represented by $\langle \Pi', C'_\Pi, S'_\Pi \rangle$ is minimal if and only if there is no other subsystem *sys*$''$ represented by $\langle \Pi'', C''_\Pi, S''_\Pi \rangle$ of *sys* such that $|\Pi''| < |\Pi'|$ or $|C''_\Pi| < |C'_\Pi|$.

The following theorem generalizes the lower bound on the number of rounds:

Theorem 4.2 *Let* sys $= \langle \Pi, C_\Pi, S_\Pi \rangle$ *be a synchronous system,* sys$' = \langle \Pi', C'_\Pi, S'_\Pi \rangle$ *be a minimal subsystem of* sys*, $\mathcal{A}$ be a Consensus algorithm, and $\kappa = |\Pi'| - \min\{|s| : s \in S'_\Pi\}$. There are two cases to be considered:*

i. If $|\Pi| - \kappa > 1$, then there is an execution of $\mathcal{A}$ in which $f \leq \kappa$ processes are faulty and some correct process takes at least $f + 1$ rounds to decide;

ii. If $|\Pi| - \kappa = 1$, then there is an execution of $\mathcal{A}$ in which $f \leq \kappa$ processes are faulty and some correct process takes at least $\min(\kappa, f + 1)$ rounds to decide.

$\square_{4.2}$

To illustrate the utilization of this theorem, consider a system *sys* $= \langle \Pi, C_\Pi, S_\Pi \rangle$ under the t of n assumption, in which processes fail by crashing. If we assume that $|\Pi| = n \geq t+2$, then $|C_\Pi| \geq 2$ and every core has size $t+1$. A minimal subsystem represented by *sys*$' = \langle \Pi', C'_\Pi, S'_\Pi \rangle$ has $n' = |\Pi'| = t+1$, $|C'_\Pi| = 1$, and $|S'_\Pi| = t+1$ (each survivor set $s \in S'_\Pi$ contains a single process). From Theorem 4.2(i), for every Consensus algorithm $\mathcal{A}$, there is an execution with $f \leq \kappa$ failures in which no process decides before round $f + 1$. The value of κ is $|\Pi'| - \min\{|s| : s \in S'_\Pi\} = t+1-1 = t$. This result matches the one given by theorem 3.2 in [3]. If we instead assume that $|\Pi| = t + 1$, then C_Π

contains a single core and *sys* is already minimal. By Theorem 4.2(ii), we have that $\kappa = |\Pi| - 1$. For some execution α of $\mathcal{A}$ with $f \leq \kappa$ failures, there is some correct process that does not decide earlier than round $\min(|\Pi| - 1, f + 1)$.

We use Theorem 4.2 in Sections 5 and 6 to derive lower bounds on the number of rounds for the crash and arbitrary failures respectively.

5 Consensus for Crash Failures

Consensus in a synchronous system with crash process failures is solvable for any number of failures [3]. 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 [20]. 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 configuration $\langle \Pi, C_\Pi, S_\Pi \rangle$:

Property 5.1 $C_\Pi \neq \emptyset$. $\Box_{5.1}$

Property 5.1 implies that there is at least one correct process in every execution. A core is hence a minimal subsystem in which Consensus is solved. Consider a synchronous system with crash failures *sys* $= \langle \Pi, C_\Pi, S_\Pi \rangle$, and a subsystem *sys'* $= \langle \Pi', C'_\Pi, S'_\Pi \rangle$ of *sys* such that $\Pi' = c_{min}$, $c_{min} \in C_\Pi$ and $(\forall c' \in C_\Pi, |c_{min}| \leq |c'|)$. By definition, *sys'* is minimal. From Theorem 4.2(ii), if $|\Pi| = |\Pi'|$, then there is some execution with $f \leq |\Pi| - 1$ process failures such that a correct process does not decide earlier than round $\min(\kappa, f + 1)$. On the other hand, if *sys* is not minimal, then there is some execution with $f \leq |\Pi'| - 1$ process failures such that a correct process does not decide earlier than round $f + 1$ by Theorem 4.2(i).

We now describe an algorithm for a synchronous system $\langle \Pi, C_\Pi, S_\Pi \rangle$, assuming that Property 5.1 holds for this system. A pseudocode and a detailed proof of correctness is provided in [8].

Our algorithm SyncCrash is based on the early-deciding algorithms discussed by Charron-Bost and Schiper [3] and by Lamport and Fischer [12]. Algorithms that take the actual number of failures into account 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 [3] is that there is a fundamental difference between early-deciding and early-stopping algorithms for Consensus. In an early-deciding algorithm, a process may be ready to decide, but may not be ready to halt, whereas an early-stopping algorithm is concerned about the round in which a process is ready to halt. A consequence of this difference, which was already noted in Section 4, is that the lower bounds for deciding and for stopping are not the same.

In an execution of SyncCrash, every process p_i begins with a proposed value v_i and an array to store the proposed values of the processes of a chosen core c. If p_i is in c, then its array initially contains only v_i, otherwise it is initially empty. A process p_i learns proposed values through messages from processes in c: in a round, every process in c broadcasts its array of proposed values to all processes in Π. Processes in $\Pi - c$, on the other hand, only receive messages. If a process $p_i \in \Pi$ receives a message containing a value v_j proposed by process p_j and p_i has no proposed value for p_j, then p_i updates position j of its array with v_j. Processes in c from which a message is received at round r, but from which no message is received at round $r + 1$, are known to have crashed before sending all the messages of round $r + 1$. This observation is used to detect a round in which no process in c crashes. Process $p_i \in \Pi$ keeps track of the processes in c that have crashed in a round, and as soon as p_i detects a round with no crashes, p_i can decide. An important observation is that when such a round r with no crashes happens (by assumption it eventually happens), all alive processes are guaranteed to have the same array of proposed values. Once a process p_i in c decides, it broadcasts a message announcing its decision value d_i. All undecided processes receiving this message decide on d_i.

If $c = \Pi$, then in every execution of SyncCrash with f process crashes, every correct process decides in at most $\min(|c| - 1, f + 1)$ rounds. Otherwise, every correct process decides in at most $f + 1$ rounds. Thus, the lower bound on the number of rounds discussed in Section 4 is tight for crash failures. Because processes in c broadcast at most one message in every round to all the processes in $|\Pi|$, the message complexity for an execution with f process crashes is $O(f * |c| * |\Pi|)$. This is, in general, better than the algorithms in [3, 12], designed with the t of n failure assumption, which have message complexity $O(f * |\Pi|^2)$.

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 [6]. Their failure model, however, assumes t of n failures, and consequently the subset used to reach agreement is not chosen based on information about correlated failures.

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 c, only three rounds are necessary in the worst case. In addition, no messages are broadcast by the processes in $\Pi - c$. This differs in most algorithms designed under the t of n assumption [3, 4, 12], although the same idea can be applied by having only a specific subset of $t + 1$ processes broadcasting messages. In this latter case, however, the subset is not necessarily minimal.

6 Consensus for Arbitrary Failures

Given a system configuration $\langle \Pi, C_\Pi, S_\Pi \rangle$, consider the following properties:

Property 6.1 (**Byzantine Partition**) *For every partition* (A, B, C) *of* Π*, at least one of* A*,* B*, and* C *contain a core.* $\square_{6.1}$

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

The following theorem states that these two properties are equivalent.

Theorem 6.3 *Byzantine Partition* $\equiv$ *Byzantine Intersection.* $\square_{6.3}$

Proof sketch:

- Byzantine Partition $\Rightarrow$ Byzantine Intersection.

We prove the contrapositive. Assume that there are two survivor sets $s_i, s_j \in S_\Pi$ such that $(s_i \cap s_j)$ does not contain a core. Consider the following partitioning: $A = \Pi - s_i$, $B = (s_i \cap s_j)$, 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. Because C contains no elements from s_j, we have that C does not contain a core. Thus, none of A, B, or C contain a core.

- Byzantine Intersection $\Rightarrow$ Byzantine Partition.

To prove this relation, we make use of two observations. First, if Byzantine Intersection holds, then every survivor set s contains at least one core. Otherwise the intersection between s and some other survivor set $s' \in S_\Pi$, $s \neq s'$, cannot contain a core. Second, if a subset A of processes contains at least one element from every survivor set, then A contains a core: by definition, in every execution there is at least one survivor set that contains only correct processes. If A contains at least one element from every survivor set, then in every execution there is at least one correct process in A.

We now assume that Byzantine Intersection holds and there is a partition (A, B, C) such that none of A, B, and C contain a core. If none of these subsets contains a core, then none of them contains neither a survivor set nor one element from each survivor set $s' \in S_\Pi$. Thus, there has to be two distinct survivor sets s_1 and s_2 such that there are no elements of s_1 in C and no elements of s_2 in B. If there are no such s_1 or s_2, then one of two possibilities has to take place, both in which at least one subset contains a core: 1) $s_1 = s_2$. In this case, A contains s_1; 2) $(\forall s \in S_\Pi, (s \cap B) \neq \emptyset) \vee (\forall s \in S_\Pi, (s \cap C) \neq \emptyset)$.

Assuming therefore that there are such survivor sets s_1 and s_2, we have that $(s_1 \cap s_2) \subseteq A$. By assumption, A does not contain a core, and consequently $s_1 \cap s_2$ does not contain a core. This contradicts, however, our assumption that Byzantine Intersection holds. $\square_{6.3}$

The utility for having two equivalent properties becomes clear below. We use Byzantine Partition to show that this replication requirement is necessary to solve Consensus in a synchronous arbitrary failure system. Byzantine intersection is assumed by our algorithm SyncByz.

Byzantine Intersection along with the definition of S_Π is equivalent to the replication requirement for blocking writes in Byzantine quorum systems identified by Martin *et al.* They show that this requirement is sufficient for such an algorithm [16]. Both our requirement and the one identified by Martin *et al.* are strictly weaker than the replication requirement for *masking quorum systems* [14]. A masking quorum system requires that in every execution at least one quorum contains only correct processes (that is, it contains a survivor set). In addition, for every quorums $\mathcal{Q}_1$, $\mathcal{Q}_2$ in a given masking quorum system and every pair of failure scenarios, there is at least one process in $\mathcal{Q}_1 \cap \mathcal{Q}_2$ that is not faulty in both scenarios. In a system satisfying Byzantine Intersection/Byzantine Partition, neither the set of cores nor the set of survivor sets necessarily satisfy the properties that define a masking quorum system. Example 6.4 illustrates such a system.

6.1 Process Replication

Byzantine Partition is necessary to solve Strong Consensus in a synchronous system with arbitrary process failures. The informal proof we provide here is based upon the one by Lamport for the t of n assumption [13, 17]. We show that, for any algorithm $\mathcal{A}$, if there is a partition of the processes into three non-empty subsets such that none of them contain a core, then there is at least one run in which agreement is violated. This is illustrated in Figure 1, where we assume the converse and consider three executions α, β, and γ.

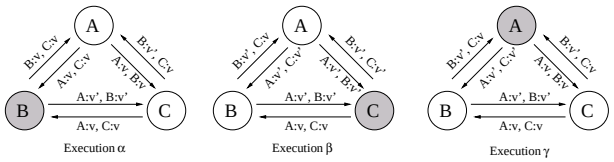

Figure 1. Executions illustrating the violation of Consensus. The processes in shaded subsets are all faulty in the given execution.

In execution α, the initial value of every process is the same, say v. All the processes in subset B are faulty, and they all lie to the processes in subset C about their initial values and the values received from processes in A. By Strong Validity, running algorithm $\mathcal{A}$ in such an execution

results in all the processes in subset C deciding v. Execution β is analogous to execution α, but instead of every process beginning with a initial value v, they all have initial value $v' \neq v$. Again, by Strong Validity, all processes in B decide v'. 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 they do in execution α. For processes in C, however, processes in B behave as they do in execution β. Because processes in C cannot distinguish executions α from γ, processes in C must decide v. At the same time, processes in B cannot distinguish executions β from γ, and therefore they must decide v'. Consequently, there are correct processes which decide differently in execution γ, violating the Agreement property.

6.2 Number of Rounds

In a synchronous system with crash failures it suffices to have a single core to solve Consensus. In general, this is not the case for synchronous systems with arbitrary process failures. The only particular case in which Consensus can be solved with a single core is the case that the system has a single reliable process p_i that does not fail in any execution. For such a system, $\langle\{p_i\},\{\{p_i\}\},\{\{p_i\}\}\rangle$ is a minimal subsystem under Byzantine Partition. In every other case, a system has to contain multiple cores. Although fault-tolerant systems may rely upon a single reliable process, this is a special case.

Assuming a minimal subsystem $\langle\Pi', C'_\Pi, S'_\Pi\rangle$ under Byzantine Partition with multiple cores, every survivor set for such a subsystem contains at least 2 processes. Otherwise, there is a core containing a single process, and it reduces to the particular case described above. By Theorem 4.2(i), the minimum number of rounds required in the worst case is $\kappa+1$, where κ is defined as $|\Pi'|-\min\{|s| : s \in S'_\Pi\}$. In contrast, all survivor sets of a minimal subsystem contain a single process, assuming crash failures.

To illustrate the difference on the total number of rounds in the worst case between the crash and the arbitrary models, consider the following example:

Example 6.4 :

- $\Pi = \{p_a, p_b, p_c, p_d, p_e\}$
- $C_\Pi = \{\{p_a, p_b, p_c\}, \{p_a, p_d\}, \{p_a, p_e\}, \{p_b, p_d\}, \{p_b, p_e\}, \{p_c, p_d\}, \{p_c, p_e\}, \{p_d, p_e\}\}$
- $S_\Pi = \{\{p_a, p_b, p_c, p_d\}, \{p_a, p_b, p_c, p_e\}, \{p_a, p_d, p_e\}, \{p_b, p_d, p_e\}, \{p_c, p_d, p_e\}\}$

$\square_{6.4}$

For crash failures, a minimal subsystem $\langle\Pi', C'_\Pi, S'_\Pi\rangle$ is such that $|\Pi|' = 2$, $|C'_\Pi| = 1$, and a minimum-sized survivor set contains a single process. By Theorem 4.2(i), the lower bound on the number of rounds is two in the worst case ($\kappa = 1$ and $|\Pi| - \kappa > 1$). For arbitrary failures, $\langle\Pi, C_\Pi, S_\Pi\rangle$ is already a minimal subsystem: if any process or core is removed, then the remaining subsystem does not satisfy Byzantine Partition. By Theorem 4.2(i), the lower bound on the number of rounds is three in the worst case ($\kappa = 2$ and $|\Pi| - \kappa > 1$). Thus, for the same system configuration, fewer rounds are required assuming crash failures.

6.3 Solving Strong Consensus

We describe an algorithm that solves Strong Consensus in a system *sys*$= \langle\Pi, C_\Pi, S_\Pi\rangle$ that satisfies Byzantine Intersection. This algorithm, called SyncByz, 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 arbitrarily faulty processes [13].

In our algorithm, all the processes execute the same sequential code. Every process creates a tree in which each node is labeled with a string w of distinct process identifiers and in which it stores a value. The value stored in a node labeled w corresponds to the value forwarded by the sequence of processes named in w. At round $r + 1$, every correct process p_j sends a message containing the labels and values of the nodes stored at depth r of the tree to all the other processes. Every correct process p_i that receives such a message stores the values contained in it in the following manner: if there is a node labeled wj, with $w \in Pid^*, |wj| = r + 1$, then store at this node the value in the message sent by p_j corresponding to w.

A simple example will help to clarify the use of the tree. Suppose that a correct process p_i receives at round three a message from process p_j that contains the string *lk* and the value v associated to this string. Process p_i stores the value v at the node labeled *lkj* and forward at round four a message containing the pair $\langle lkj, v\rangle$ to all the other processes.

The leaves in this tree are survivor sets. More specifically, if we use *Node*(w) to denote the node of the tree labeled with the string w and *Processes*(w) the set of processes named in w, then *Node*(w) is a leaf if and only if $\Pi - $*Processes*($w$) does not contain a survivor set. Consequently, if *Node*(wi) is a leaf and we denote with *Child*(w) the set of processes $\{p_i|$*Node*(*wi*) is a child of *Node*(*w*)$\}$, then *Child*(w) is a survivor set[2]. For every non-leaf *Node*(w), we have that $\Pi -$ *Processes*(w) has to contain a survivor set. A consequence of this definition is that the depth of the tree is $|\Pi| - \min\{|s_i| : s_i \in S_\Pi\} + 1$. Figure 2 gives an example of a tree for the system configuration in Example 6.4.

The first stage of the algorithm builds and initializes the tree. The second stage runs several rounds of message exchange. In the first round, each process broadcasts

[2]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.

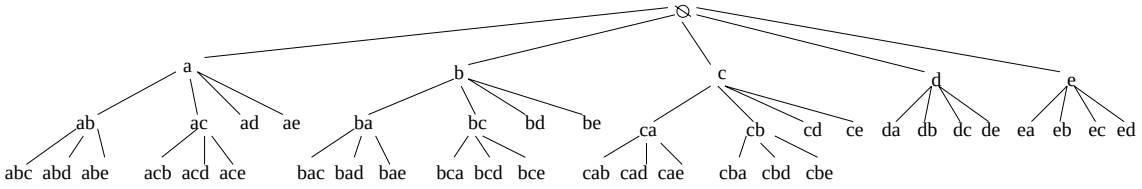

Figure 2. An example of a tree built by each process in the first stage of the algorithm.

its initial value, and in subsequent rounds, each process broadcasts the values it learned in the previous round. As processes receive the messages containing values learned in previous rounds, each node populates 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. Finally, in the last round, each process traverses the tree visiting the nodes in postorder to decide upon a value. When visiting a leaf, the algorithm does not change the value this node stores. On the other hand, when an internal node of process p_i with label w is visited, we use a replacement strategy to determine its value. Suppose there are two survivor sets s_1 and s_2 such that $(s_1 \cap s_2) \subseteq \mathit{Child}(w)$ and for every process $p_j \in (s_1 \cap s_2)$, we have that $\mathit{Value}(wj) = v$, for some $v \in (V \cup \{\perp\})$. In this case, we replace the value of *Node*(w) with v. Otherwise we replace with the default value ($\perp$). In the original algorithm, the replacement strategy is based on the majority [1].

Instead of providing a formal proof of correctness for SyncByz, we illustrate the decision process for the system described in Example 6.4. For a proof of correctness and a pseudocode, we point the interested reader to [8].

After $|\Pi| - |s_{min}| + 1 = 5 - 3 + 1 = 3$ rounds of message exchange, every correct process has populated its tree with values received from other processes. The values stored at non-leaf nodes are not important, because they are replaced according to the strategy defined for the algorithm during the traversal of the tree. We illustrate this procedure for the subtrees rooted at both *Node*(a) and *Node*(b). This is shown in Figures 3 and 4. White nodes are the ones that have the same value across all the correct processes, whereas shaded nodes are the ones that have possibly different values across correct processes. A node is shaded if the last node in the string that labels the node is a faulty process. Note that if two nodes *Node*(w) and *Node*(w') are white, it does not mean that they contain necessarily the same value. It only means that every correct process has value v at node w and v' at node w'.

Consider the particular scenario in which processes p_a and p_c are faulty and p_b, p_d, and p_e are all correct. First, we discuss the subtree rooted at *Node*(a). At Time 1, only the nodes at the last level have been visited. From the algorithm, when a leaf is visited, its value does not change. Thus, the state of the tree at Time 1 is the same state as right before starting the traversal of the tree. Time 2 corresponds to the state of the tree exactly after all the nodes at Level 2 are visited. Because processes p_b, p_d, and p_e are correct, *Node*(abd) and *Node*(abe) contain the same value across all correct processes. By the replacement strategy of the algorithm, the new value of node ab is the value of nodes abd and abe, because $\{p_d, p_e\} \subseteq \{p_a, p_d, p_e\} \cap \{p_b, p_d, p_e\}$ and *Node*(abd) contains the same value as node abe. Similarly, the new value of node ac is the one of acb, acd, and ace. The values of *Node*(ad) and *Node*(ae) across correct processes have to be the same, because p_d and p_e are correct. At Time 3, the value of *Node*(a) become the same for all correct processes. Since the value of *Node*(ab), *Node*(ac), *Node*(ad), and *Node*(ae) are the same across all correct processes, the new value of node a has to be the same.

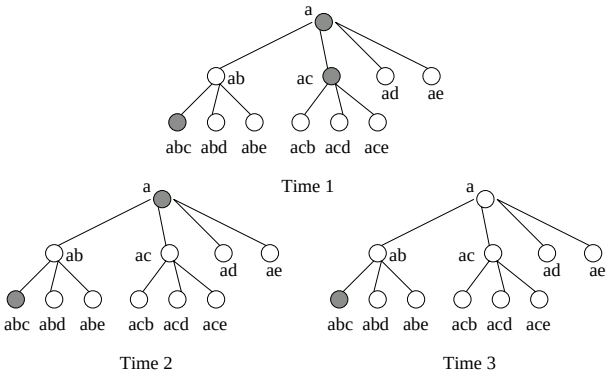

Figure 3. An example of traversing the subtree rooted at $\mathit{Node}(a)$. Time i corresponds to the state of the tree exactly after all the nodes of Level $4 - i$ are visited.

For the subtree rooted at *Node*(b), the value of *Node*(ba) may still not be the same across all correct processes at Time 2. Both $\{p_d, p_e\}$ and $\{p_c, p_d\}$ are cores and are subsets of processes contained in some intersection of two survivor sets. Thus, if the value of *Node*(bac) is the same of *Node*(bad) in a correct process p_i, but different in another correct process p_j, then p_i and p_j may replace the value of *Node*(ba) with different values, depending on *Node*(bae). Note that one value must be the default $\perp$ and the other some $v \in V$. Similarly for *Node*(bc) at Time 2. The values

of *Nodes*(bd) and *Nodes*(be), however, have to be the same across all correct processes. In Time 3, $\{p_d, p_e\}$ is the only core in *Child*(b) to contain the same value in their respective nodes at Level 1, unless ba and bc have the same value as *Node*(bd) and *Node*(be). Furthermore, this core is in the intersection of $\{p_b, p_d, p_e\}$ and $\{p_a, p_d, p_e\}$. Consequently, the new value of *Node*(b) has to be the same for every correct process at Time 3, by the value replacement strategy.

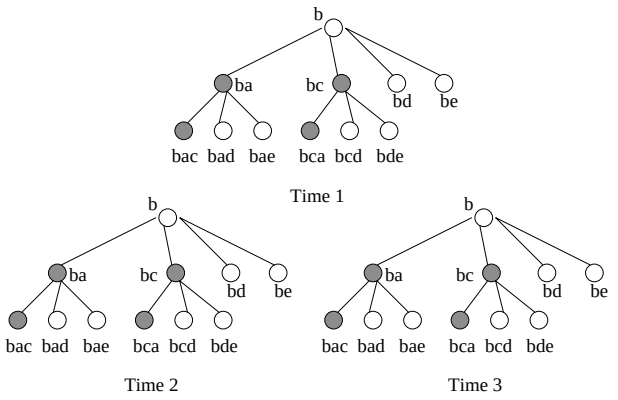

Figure 4. An example of traversing the subtree rooted at *Node*(b). Time i corresponds to the state of the tree exactly after all the nodes of Level $4 - i$ are visited.

By doing the same analysis for the subtrees rooted at nodes c, d, and e, we observe that every node at Level 1 of the tree rooted at $\emptyset$ has the same value across all correct processes. Therefore, the decision value, which is the value at node $\emptyset$ after visiting it, has to be the same for every correct process. One important observation is that the value at *Node*(i) across all correct processes is the initial value of process p_i, if p_i is correct. In the case that every process has the same initial value v, then the decision value has to be v.

To illustrate the benefits of using our abstraction, consider once more the five process system of Example 6.4. If the t of n failure assumption is used, then Strong Consensus is not solvable: the smallest survivor set contains three processes and so the maximum number of failures in any execution is two. With the t of n assumption, the replication requirement is $|\Pi| \geq 3t + 1$, and for $t = 2$, it is necessary to have at least seven processes. With our model, however, algorithm SyncByz solves Strong Consensus.

7 Practical Considerations

Two important issues concerning the use of cores and survivor sets are (1) how to extract information about these subsets and (2) how to represent these subsets.

To extract core information (such as finding a smallest core) using failure probabilities is an NP-hard problem in the general case [10]. This result need not be discouraging, however. First, this is a problem that is already addressed for many safety critical systems, and techniques have been developed to extract such information [18]. Furthermore, for many real systems, there are simple heuristics for finding cores that depend on how failure correlations are identified. Suppose a system in which processes are tagged with colors. In this model, all processes have identical probabilities of failing, but those with the same color have highly-correlated failure probabilities while processes with different colors fail independently. A core in such a system is composed of processes with different colors, and the size of a core depends on the probability of having colors failing. To find cores in such a model, one has to interactively add processes with different colors to a subset and verify whether this subset is a core. The verification procedure consists in multiplying the probability of failure for every color that has a representative in the subset. This clearly can be accomplished in polynomial time. For real systems, a color would represent some intrinsic characteristic. For example, all components in a certain part of an airplane are likely to fail if there is a structural damage on that particular part. Computers in the same room are subject to correlated crash failures in the case of a power outage.

One can go further in extracting cores based on characteristics of the system and propose the utilization of several attributes, instead of one as in the color model. It turns out that in the general case, this problem is also NP-hard. Some simplifying assumptions such as finding orthogonal cores (cores in which processes do not share attributes) make the problem tractable. Finally, fault tree analysis is an option in the design of reliable systems.

Representing cores or survivor sets is relevant for arbitrary failures. As discussed in Section 5, to solve Consensus assuming crash failures, it suffices a single core. Space complexity is hence $O(n)$ in this case. For arbitrary failures, however, multiple survivor sets are necessary in general. An important observation is that the number of processes in fault-tolerant systems is usually not large. Thus, for a small number of processes, the number of survivor sets is perhaps suitable for an implementation, even if it is exponential on the number of processes. Alternatively, there are cases in which it is not necessary to store information about all survivor sets, because it is possible to determine algorithmically whether a subset of processes is a survivor set. Considering the color model once more, a subset s of processes is a survivor set if $\Pi - s$ does not contain a core. Whether $\Pi - s$ contains a core or not is verifiable in polynomial time, as discussed previously.

8 Conclusions

Cores and survivor sets are abstractions that enable the design of distributed algorithms for dependent process failures. In this paper, we have used such abstractions to derive

process replication requirements for the Consensus problem in the synchronous model. Although we have considered only Consensus, we conjecture that these are also requirements for a broader set of problems in fault-tolerant distributed computing. One of our current goals is to show this conjecture.

Regarding the practical application of these abstractions, we have been working with real scenarios that require the utilization of dependent failure models. In particular, we have been targeting Internet catastrophes (e.g. Internet worms, viruses, etc). For these scenarios, the t of n assumption is not adequate, because the number of hosts that may fail dependently is potentially very large. This is therefore an example in which models for dependent failures are clearly necessary. We expect to use our experience in these settings to confirm that our model is useful for the design of distributed systems.

Acknowledgments

This work was directly and indirectly influenced by the interaction with several people. In particular, we would like to thank Fred B. Schneider, John Knight, Stefan Savage, Geoffrey M. Voelker, André Barroso, Renata Teixeira, Dmitrii Zagorodnov, Xianan Zhang, and the people of RAMP (John Bellardo, Ranjita Bhagwan, Marvin McNett, Alper Mizrak, and Jessica Wang) for their support and comments.

References

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

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

[3] B. Charron-Bost and A. Schiper. Uniform Consensus is Harder Than Consensus. Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[4] D. Dolev, R. Reischuk, and H. R. Strong. Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

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

[6] 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)*, pages 168–177, Sendai, Japan, June 1996.

[7] M. Hirt and U. Maurer. Complete Characterization of Adversaries Tolerable in Secure Multi-Party Computation. In *ACM Symposium on Principles of Distributed Computing*, pages 25–34, Santa Barbara, California, 1997.

[8] F. Junqueira and K. Marzullo. Consensus for Dependent Process Failures. Technical Report CS2003-0737, UCSD, La Jolla, CA, September 2002.

[9] F. Junqueira and K. Marzullo. Lower Bound on the Number of Rounds for Synchronous Consensus with Dependent Process Failures. Technical Report CS2003-0734, UCSD, La Jolla, CA, September 2002.

[10] F. Junqueira, K. Marzullo, and G. Voelker. Coping with Dependent Process Failures. Technical Report CS2002-0723, UCSD, La Jolla, CA, December 2001.

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

[12] L. Lamport and M. Fischer. Byzantine Generals and Transaction Commit Protocols. Available at URL http://research.microsoft.com/users/lamport/pubs/trans.pdf, April 1982.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[14] D. Malkhi and M. Reiter. Byzantine Quorum Systems. In *29th ACM Symposium on Theory of Computing*, pages 569–578, may 1997.

[15] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *10th Computer Security Foundations Workshop (CSFW97)* , pages 116–124, Rockport, MA, June 1997.

[16] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine Storage. In *16th International Symposium on Distributed Computing (DISC 2002)*, Toulouse, France, October 2002.

[17] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[18] Y. Ren and J. B. Dugan. Optimal Design of Reliable Systems Using Static and Dynamic Fault Trees. *IEEE Transactions on Reliability*, 47(3):234–244, December 1998.

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

[20] D. Skeen. Determining the Last Process to Fail. *ACM Transactions on Computer Systems*, 3(1):15–30, February 1985.

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

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

[23] J. H. Wensley, M. W. Green, K. N. Levitt, and R. E. Shostak. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. In *2nd IEEE International Conference on Software Engineering*, pages 458–469, 1976.