A New Algorithm to Solve Synchronous Consensus
for Dependent Failures

Jun Wang, Min Song
Electrical and Computer Engineering Department
Old Dominion University

Abstract

Fault tolerant algorithms are often designed under the \( t \)-out-of-\( n \) assumption, which is based on the assumption that all processes or components fail independently with equal probability. However, real systems may exhibit dependent failures. Cores and survivor sets are used to build an abstraction model for dependent process failures. Using this abstraction, we design an algorithm to solve consensus problem for system configurations that is different and in which is not possible to solve under the \( t \)-out-of-\( n \) assumption. In consensus problem, each process starts with a proposed initial value, and finally all non-faulty processes decide on the same value. Our algorithm uses the processes in all cores to broadcast messages. Each core reaches agreement separately and even simultaneously in some round with no failure in the core. In the worst-case, the decision can be made in the round that is equal to the size of the minimal core. It is found that different cores may decide on different values, if the processes have different initial values. However, our algorithm guarantees that all processes eventually decide on the same value regardless the initial values. We prove the correctness of our algorithm and give the lower bounds of the number of rounds to solve consensus problem.

Keywords: Distributed system, synchronous consensus, dependent failures

1. Introduction

Consensus problem in the presence of failures is a fundamental issue of both practical and theoretical importance. It consists in reaching agreement among a set of processes upon a value. Uniform consensus is a stronger problem in the sense that it requires that no two processes decide distinct values. In the crash failure model, consensus is often specified in terms of the following three properties: Validity, Agreement, and Termination.

Most fault tolerant algorithms found in the literature are designed under the assumption that no more than \( t \)-out-of-\( n \) processes. This assumption is most useful when components have identical probabilities of failure and they fail independently. However, in real systems, failures can have different probabilities and often be highly correlated. To model dependent process failures Junqueira and Marzullo introduced two types of sets, cores and survivor sets [1]. Cores and survivor sets enable the design of fault tolerant algorithms for systems in which components do not have independent and identically-distributed probabilities of failure.
In this paper, we extend the work to solve synchronous consensus for dependent crashed failures. Our algorithm uses all processes in all cores instead of in a single core to broadcast message. Each core decides a value independently. Once a decision is made, the core broadcasts the decision to all other undecided processes. For each individual core to decide, we define the partially identical vectors of the core. While several cores decide in the same round, if the processes have the same initial value, each core must decide on the same value. On the other hand, if the processes have different initial values, each core may decide on different values. In this case, our algorithm uses the weight of the core to break the decision tie.

The rest of the paper is organized as follows. Section 2 briefly surveys the related work. In Section 3, we present the background of our work, including the system model, the abstraction that models dependent process failures, and the consensus problem. In Section 4, we propose our algorithm for synchronous consensus on the crash failure, and prove the correctness of the algorithm. Section 5 concludes the paper.

2. Related Works

The \( t \)-out-of-\( n \) assumption is implicitly based on the following assumptions [3]: (1) all failure types are equally likely; (2) the probability of a component failing while a protocol is in progress is independent of the duration of the protocol; (3) all components that can fail have an identical probability of failure; and (4) failure probabilities of different components are mutually independent. For example, probabilistic models generally assume that processes fail after an exponentially distributed random time, independently of each other [8]. Therefore, process has a fixed failure probability in a given step if each step runs in an equal duration. With this assumption, it is fairly easy to design and verify protocols and also to express lower and upper bounds for the number of rounds to solve consensus. If processes fail in an arbitrary manner, without using digital signatures, the consensus problem of synchronous system is solvable when \( n > 3 \times t \) [2]. However, these assumptions do not adequately reflect the nature of real-world network environments. Different failures can have different probabilities of occurring. For different failure modes of the same component, assumption (1) is relaxed by hybrid failure models [4], which provide separate \( t \)'s for different classes of failures, e.g., crash failures and arbitrary failures. Assumptions (3) and (4) are rarely questioned; even hybrid failure models still make assumptions about the independence of failures within each class. Parvedy and Raynal used this assumption to solve uniform consensus with crashed and process omission failure model, such that every process that decides, including faulty process, must decide upon the same value [7].

Junqueira and Marzullo studied various distributed problems in the context of dependent process failures, and introduced the concepts of cores and survivor sets to model dependent process failures [1]. The core is defined as the minimal subset of processes such that at least one is correct in any execution, and a survivor set is a minimum subset of processes that contain the correct processes of some execution. Their model translates the subsets of size \( t + 1 \) and \( n - t \) in the \( t \)-out-of-\( n \) assumption into cores and survivor sets, individually. According to the \( t \)-out-of-\( n \) assumption, arbitrary \( t + 1 \) processes have at least one correct process in any execution, which is similar as the definition of the core. Using the abstraction of cores and survivor sets, they proposed algorithms for consensus on the crash and arbitrary failure models, and generalized the lower bound on the number of rounds to solve consensus [1, 6]. Their algorithm of consensus for
3. System Model and Failure Abstraction

3.1 System Model

We consider a distributed system consisted of $n$ processes. Each process $p_i$ initially proposes a value $v_i$. A process communicates with others by exchanging messages through point-to-point reliable channels. In this paper, processes are assumed not to be reliable. We consider crashed process failures, in which processes fail by crashing, and failed processes do not recover. Therefore, if a process is faulty in an execution, then it prematurely stops sending and receiving messages in that execution. 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 $t' < t$. Moreover, process failures may be correlated, which means that the failure of a process may indicate an increase in the failure probability of another process.

We assume the system is synchronous, which imposes upper bounds on processing time, message transfer delay, and clock drift. In a synchronous system, an execution of an algorithm can be further organized in successive rounds [5]. During each round, the correct processes exchange messages, and stop and decide when they have reached a round such that they all have converged to the same decision value.

3.2 Execution

An execution $\phi$ of an algorithm $A$ is a sequence of steps of the processes in $\Pi$, defined as a tuple $\langle F_\phi, I_\phi, S_\phi, T_\phi \rangle$ [1]. $F_\phi$ evaluates to the subset of processes that have failed by time $t$. $I_\phi$ indicates the initial configuration of the processes. $S_\phi$ is an infinite sequence of steps of the processes in $\Pi$. The time $t$ at which a step $e \in S_\phi$ is executed is given by $T_\phi(e)$. A difference of them may change the result of the computation.

The faulty behavior of a crashed process is observable as soon as it crashes. From the definition of an execution, we define the set of correct process in an execution $\phi$ as $Correct_\phi = \Pi - \bigcup_{t \in T_\phi} F(t)$, and the set of failed processes as $Faulty_\phi = \bigcup_{t \in T_\phi} F(t)$.

Additionally, we define the subset of crashed processes that failed by round $r$ as $Crashed_\phi$. A process $p_i$ is in $Crashed_\phi$ if it has not executed all the steps of some round $r' \leq r$.

3.3 Cores and Survivor Sets

We restate the definition of core and survivor set in [1]. Let $\Pi$ be the set of processes under consideration, $\Phi$ be the set of all valid executions, and $Correct_\phi$ be the set of correct
processes of execution \( \phi \in \Phi \).

**Definition 1**: A subset \( c \subseteq \Pi \) is a core if and only if: 1) \( \forall \phi \in \Phi, \ Correct_{\phi} \cap c \neq \phi \); 2) \( \forall p_i \in c, \ \exists \phi \in \Phi \) such that \( Correct_{\phi} \cap (c - \{p_i\}) = \phi \).

**Definition 2**: A subset \( s \subseteq \Pi \) is a survivor set if and only if: 1) \( \exists \phi \in \Phi, \ Correct_{\phi} = s \); 2) \( \forall \phi \in \Phi \) and \( (s - \{p_i\}) \not\subset Correct_{\phi} \).

In general, a core corresponds to a reliable subset of processes, and is a non-empty and minimal subset of processes such that at least one process is correct in every execution. A survivor set is a minimum subset of processes that contain the correct processes of some execution. So in every execution of the system, there exists at least one correct process in each core, and at least one survivor set such that it contains only correct processes. Donate \( C_{\Pi} \) the set of all cores of \( \Pi \), \( S_{\Pi} \) the set of all survivor sets of \( \Pi \), and use a triple \( \langle \Pi, C_{\Pi}, S_{\Pi} \rangle \) to represent a system configuration: the set of processes that compose the system along with its cores and survivor sets. \( C_{\Pi} \neq \phi \), such that there is at least one correct process in any execution. \( S_{\Pi} \) can be directly determined from \( C_{\Pi} \), as every survivor set is the minimal subset that has a non-empty intersection with every core. Assume that \( C_{\Pi} \) and \( S_{\Pi} \) are static, meaning that there are no modifications in the membership of processes or in the cores/survivor sets of the system under consideration.

### 3.4 Consensus

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. We assume that the default value \( \bot \) is not in \( V \), and \( V \cup \bot \) denote that a decision value is either in \( V \) or the default. In crash failure model, Consensus is specified in terms of the following three properties:

**Validity**: If some non-faulty process \( p_i \in \Pi \) decides on value \( v \), then \( v \) was proposed by some process \( p_j \in \Pi \);

**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 on a value.
4. A New Algorithm for Crash Failures

Synchronous consensus with crash process failures is solvable for any number of failures. Before giving the algorithm to solve synchronous consensus problem, we first consider the following definitions.

**Definition 3**: The union of all cores \[ C = \bigcup_k c_k, \quad \forall c_k \in \Pi. \]

Our algorithm on crash failure is based on the processes in \( C \). Let SynCrash be the algorithm in the remainder of this paper. If we assume that \( \Pi \) is the complete sets of all cores in \( \Pi \), then \( \Pi - C \) is the set of processes that are not in any core, which actually has no effect on fault tolerance of the system. If \( C = \Pi \), SynCrash uses the whole set of processes instead of a subset of processes to reach agreement. If \( C \) is not the complete sets of all cores in \( \Pi \), SynCrash uses multiple cores to broadcast messages. The algorithm enables either only one core to decide, or several cores to simultaneously decide. If only one core decides in the one round, other processes may decide in the same round by detecting the round without crashes in the core, or decide in the next round by receiving a decision message for the process in the core. In the case that several cores decide in the same round in one execution, we consider that all processes have same initial value or different initial values.

**Definition 4**: The minimal core \( c_{\min} \) is the core such that \( c_{\min} \in \Pi \), and \( \forall c_k \in \Pi \),

\[ |c_{\min}| \leq |c_k|. \]

Notice that there may exist several cores with \( |c_{\min}| \) number of processes, but \( c_{\min} \) is used to formalize the idea of the minimal core, and the algorithm can choose any one of them if the initial values of all processes are same. If not, the algorithm chooses one special minimal core to make the decisions.

Similarly, the minimal survivor set \( s_{\min} \) is the survivor set such that \( s_{\min} \in \Sigma \), and \( \forall s_k \in \Sigma \),

\[ |s_{\min}| \leq |s_k|. \]

**Definition 5**: Let \( \phi \) be an execution of SynCrash, the number of faulty processes in core \( c_i \) is \( f_i = |c_i| - |c_i \cap \text{Correct}_\phi| \). Then \( \forall c_i \in \Pi \), the minimal number of failures is \( \min_j(f_j) \), and \( 0 \leq f \leq |c_{\min}| - 1 \) is hold.

For \( c_{\min} \), since there is at least one correct process in any execution, \( \forall f_i \),

\[ 0 \leq f_i \leq |c_{\min}| - 1 \] is hold, so does \( f \).

Let \( e_i \in S_\phi \) be a step of \( p_i \) such that \( p_i \) receives its last message of round \( r \) at step \( e_i \),
\[ e_j \in S_p \] be a step of \( p_j \) such that \( p_j \) receives its last message of round \( r \) at step \( e_j \), \( e_i', e_j' \in S_p \) be any two steps of \( p_i \) and \( p_j \), respectively, in round \( r \), such that \( T(e_i') \geq T(e_j') \) and \( T(e_i') \geq T(e_j) \).

**Definition 6:** Processes \( p_i \) and \( p_j \) have partially identical vectors for some core \( c_k \) in round \( r \) if and only if for every \( p_i \in c_k \), \( v_i[I] = v_j[I] \), where \( v_i \) is the vector of proposed values of \( p_i \) after taking step \( e_i' \) and \( v_j \) is the vector of proposed values of \( p_j \) after taking step \( e_j' \).

Before presenting a pseudo-code of the algorithm, we provide a table describing the variables used in the algorithm. Table 1 describes the variables, and the pseudo-code of SynCrash is presented in Figure 1.

Table 1: Variable used in the algorithm SynCrash

| \( c_{\min} \) | Core which has minimal processes |
| \( C \) | The set of processes consists of the union of all cores |
| \( dev_i[0 \cdots K], dec_i[k] \in V \cup \{\bot\} \) | Vector of decision value for core \( c_k \); \( K \) is the number of cores, and \( dev_i[0] \) indicates the decision value for all cores |
| \( d[0 \cdots K] \in \{true, false\} \) | Boolean variable indicates whether the process decided for \( c_k \) in the previous round, \( d[0] \) indicates decision for all cores |
| \( v_i[1 \cdots |C|], v_j[j] \in V \) | Vector of proposed values |
| \( e_i[1 \cdots c_{\min} - 1], e_j[r] \subset C \) | Array of failed processes. \( e_i[r] \) stores the subset of processes detected by \( p_i \) as crashed in round \( r \) |

Suppose all processes have different initial value. The following is the algorithm SynCrash for process \( p_i \):

Input: set \( \Pi \) of processes; set \( C_\Pi \) of cores; initial value \( v_i \in V \)

Initialization:
\[
C = \bigcup_k (c_k), \forall c_k \in C_\Pi, c_{\min} \in C_\Pi
\]
\[
dec_i[1 \cdots K], dec_i[k] = \bot; d_k \leftarrow false
\]
\(v, \{1 \cdots |\mathcal{C}|\} \quad v(\mathcal{C}) = \bot, \forall k \in \{1 \cdots |\mathcal{C}|\}\) If \(p_i \in \mathcal{C}, v_i[1] \leftarrow v_i\)

\(e_i[0 \cdots |c_{\min}| - 1] e_i[0] = \phi, e_i[k] = C, \forall k \in \{1 \cdots |c_{\min}| - 1\}\)

Round \(1 \leq r < |c_{\min}|\), \(\forall p_i \in \mathcal{C}:\)

if \((d[k] = \text{false})\) then

send \((i, v_i)\) to all process \(p_j\) such that \(\exists c_k \in \mathcal{C}_i\) and \(p_i, p_j \in c_k\)

send \((i, v_i)\) to all other processes in \(\Pi\)

else

send \((\text{Decide}, \text{dec}_k)\) to all process \(p_j\) such that \(\exists c_k \in \mathcal{C}_i\) and \(p_i, p_j \in c_k\)

send \((\text{Decide}, \text{dec}_k)\) to all other processes in \(\Pi\)

upon reception of message \((\text{Decide}, \text{dec}_k)\) do

\(\text{dec}_i[k] \leftarrow \text{dec}_k\)

\(d[k] \leftarrow \text{true}\)

upon reception of message \((j, v_j)\) do

\(e_i[r] \leftarrow e_i[r] - \{j\}\)

for \(k = 1\) to \(|\mathcal{C}|\) do

\(\text{if } (v_j[k] \neq \bot) \text{ then } v_i[k] = v_j[k]\)

for \(c_k \in \mathcal{C}_i\)

\(\text{if } ((d[k] = \text{false}) \cap ((e_i[r - 1] \cap c_k) = (e_i[r] \cap c_k)) \cup ((|c_k| = |c_{\min}|) \cap (r = |c_k| - 1))))\)

\(\text{dec}_i[k] \leftarrow \min_{p_j \in c_k} (v_j[k])\)

\(d[k] \leftarrow \text{true}\)

if \((\exists k, d[k] = \text{true})\) then

\(\text{dec}_i[0] \leftarrow \min_{c_k \in \mathcal{C}_i} (\text{dec}_i[k])\)

\(d[0] \leftarrow \text{true}\)
Round \([c_{\min}], \forall p_i \in c_{\min}\)

send \((\text{Decide}, \text{dec}_k)\) to all processes in \(\Pi - c_{\min}\)

upon reception of message \((\text{Decide}, \text{dec}_k)\) do

\[\text{dec}[,k] \leftarrow \text{dec}_k\]

\[\text{dec}[,0] \leftarrow \min_{c_j \in \Pi \cap c_{\Pi}} (\text{dec}[,k])\]

halt

Round \(1 \leq r < c_{\min}, \forall p_i \in \Pi - C:\)

upon reception of message \((\text{Decide}, \text{dec}_k)\) do

\[\text{dec}[,k] \leftarrow \text{dec}_k\]

\[d[,k] \leftarrow \text{true}\]

upon reception of message \((j, v_j)\) do

\[e_i[,r] \leftarrow e_i[,r] - \{j\}\]

for \(k = 1\) to \(|C|\) do

if \((v_j[,k] \neq \perp)\) then \(v_i[,k] = v_j[,k]\)

for \(c_k \in C_{\Pi}\)

if \((e_i[,r-1] \cap c_k) = (e_i[,r] \cap c_k)\) then

\[\text{dec}[,k] \leftarrow \min_{p_j \in \Pi} (v_i[,j])\]

\[d[,k] \leftarrow \text{true}\]

if \((\exists k, d[,k] = \text{true})\) then

\[\text{dec}[,0] \leftarrow \min_{c_j \in \Pi \cap c_{\Pi}} (\text{dec}[,k])\]

\[d[,0] \leftarrow \text{true}\]

halt

Figure 1: Synchronous consensus for dependent crash failures
Our algorithm differentiates the processes of $C$ from the rest of the processes in $\Pi - C$. At each round, every process in $C$ broadcasts its knowledge of proposed values to all the other processes, whereas processes in $\Pi - C$ just listen to these messages. Processes in $C$ from which a message is received in round $r$, but from which no message is received in 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 some core crashes. Process $p_i \in \Pi$ uses an array of failed processes to keep track of the processes in $C$ that have crashed in a round, and as soon as $p_i$ detects a round with no crashes in any core, $p_i$ can decide. An important observation is that when such a round $r$ with no crashes in any core happens, all live processes are guaranteed to have the same vector of proposed values. Another observation is that even the processes in different cores decide in the same round, they decide on the same value if all processes have the same initial value. In the other case, if no such a round, by assumption there is at least one correct process in every core $C_k \in C$, the process will eventually decide in the same round. Since all processes receive the messages for the processes in the minimal core and the minimal core has one correct process, they decide in the round of $c_{\min} - 1$.

Once process $p_i \in C$ decides, it broadcasts a decision message announcing its decision value $v_i$. All undecided processes receiving this message decide on $v_i$ as well. Thus, only two types of messages are necessary in the protocol: messages containing the vector of proposed values and decision messages with the decision value. Because processes $p_i \in C$ broadcast at most one message in every round to all the processes in $\Pi$, the message complexity is $O(|C|*|\Pi|)$, which is no more than that of the algorithms for t out of n assumption, and no less than that of the algorithm in [1].

In addition, if $C = \Pi$, in every execution, every correct process decides in at most $\min[|C_{\min}| - 1, f + 1]$ rounds. Otherwise, every correct process decides in at most $f + 1$ rounds, where $f$ is the minimal number of failures in all cores that crash in a given execution $\phi$. Notice that under $t$-out-of-$n$ assumption, $f$ is the number of failures of a given execution, and is the number of failures in one particular core $c$ in a given execution. In a synchronous system with crash failures, it suffices to have a single core to solve consensus. Our algorithm uses all processes in $C$ to broadcast messages, and actually uses one of cores for the decision as soon as it can decide.

If all processes have the same initial value, our algorithm can solve the consensus problem for crash failures. If the initial values are different, while several cores decide simultaneously, they may decide on the different values. This may cause a conflict. Moreover, the protocol is round-based, but a process can fail at any step, especially at the step sending out messages. For two correct processes $p_i, p_j \in Correct_\phi$, if process $p_i \in c_k$ fails after sending the message
to $p_i$ and before sending the message to $p_j$ in round $r$, $p_i$ and $p_j$ receive different set of messages in this round, and may have different partial vector for $c_k$. If $p_i$ decides due to detecting no crashes in $c_k$ in round $r$, $p_j$ may not decide. Since the number of failures in $c_k$ is the upper bound round whether $p_j$ receives the value in the entry $i$ of the vector or receives the decision message from process in $c_k$, $p_j$ only decides for $c_k$ after it receive the value or the decision message. The decision conflict can also occur between the rounds that $p_i$ and $p_j$ decide. To guarantee that for $c_k$, $p_i$ and $p_j$ have partially identical vector and decide at the same value, $p_i$ and $p_j$ must receive the same set of messages.

While different cores may decide in the same round, we use the weight of core to break the tie. We define $C_{id}$ as the set of core ID's, i.e., $C_{id} = \{k : c_k \in C_{\Pi}\}$, where $c_k$ refer to a core with a ID of $k$. Without loss of the generality, we use core ID as the weight of core, indicating that less value $k$ of $c_k$ has more priority. We redefine $c_{min}$ is the special minimal core with the minimal core ID. Process $\forall p_i \in C$ does not send out any decision message in round 1 to $c_{min} - 1$. Instead, $p_i$ decides when it detects no crashes in some core, and then halts. It only broadcasts the decision message in round $c_{min}$ in the case that it decides when it detects there is only one correct process in $c_{min}$.

5. Conclusions

To solve synchronous consensus for dependent crashed failures, we have presented an algorithm which uses all processes in all cores instead of in a single core to broadcast messages. Multiple cores decide separately and simultaneously. For using multiple cores to decide, we define the partially identical vectors of the core. If one core satisfies the partially identical vectors, it can make a decision separately. If one core makes the decision before other cores at some round, our algorithm has the same result as the algorithm in [1]. If several cores make the decision at some round $r$, the processes in the decision cores broadcast their decisions to all other undecided processes. In this case, if processes in these cores have the same initial value, each core must decide on the same value. On the other hand, if processes have different initial values, each core may decide on the different values at round $r$. In this case, our algorithm uses the weight of the core to break the decision tie, and all processes decide at the value of the winner core at the next round, thus eventually decides on the same value.

In the worst case, our algorithm has the same bound for the number of rounds as algorithms in [1] if $c_{min}$ is chosen as the decision core. However, If there exists any core in which has $f \leq |c_{min} - 1|$ failures, that core will make the decisions at round earlier than round $c_{min}$, and the number of round to achieve the agreement is $f + 1$. In our algorithms, $f$ is defined as the minimal number of failed processes in all cores instead of the number of failed processes in a core. The maximal number of failures is $|\Pi| - |s_{min}|$. 
References


