A Synthesized Algorithm for Interactive Consistency

Adrià Gascón and Ashish Tiwari

SRI International, Menlo Park, CA 94025

Abstract. We revisit the interactive consistency problem introduced by Pease, Shostak and Lamport. We first show that their algorithm does not achieve interactive consistency if faults are transient, even if faults are non-malicious. We then present an algorithm that achieves interactive consistency in presence of non-malicious, asymmetric and transient faults, but only under an additional guaranteed delayed ack assumption. We discovered our algorithm using an automated synthesis technique that is based on bounded model checking and QBF solving. Our synthesis technique is general and simple, and it is a promising approach for synthesizing distributed algorithms.

1 Introduction

Distributed consensus is a fundamental problem in Computer Science. The goal is to reach agreement in a distributed system in the presence of faults. Depending on the formulation of the problem, it has been called the distributed agreement, distributed consensus, or interactive agreement.

Consider a distributed system composed of n processes that can communicate to each other only by means of two-party messages. Each process has a local Boolean value that it wishes to share with all other processes. Eventually we want all the processes to know each other's local value. Achieving this desired final configuration is complicated by the presence of faults.

We will assume a synchronous timinig model; that is, there are known bounds on the time required for executing one step of a process and on the time required for a message to reach its destination. Formally, we will assume that process execution times are negligible. So, the distributed agreement protocol can work in *rounds*: in each round, every process receives the messages its neighbors had sent in the previous round and it sends out a new message to each of its neighbors.

A key challenge in achieving agreement is to do so in the presence of faults. In our presentation, we assume that processes can be faulty, but the communication channel is reliable. As it will become clear to the reader later, we can also formulate our results in a way that makes the processes reliable and the channels faulty. Moreover, every pair of processes communicate through a dedicated channel and hence the receiver of a message always knows which process sent it.

So, what do we assume about the nature of a fault? First, let us define some attributes of faults. A fault is *transient* if the identity of the faulty process is not fixed; that is, a process that is faulty in the current step can become non-faulty in the next step and vice versa. In contrast, a fault is *permanent* if the same set of processes remain faulty at every synchronous step.

A fault is *benign or non-malicious or fail-silent* if every faulty processes either behaves exactly as a non-faulty process or sends just *nil* messages to other processes. Note that this kind of fault is equivalent to a message not being properly delivered or a process refusing to send a particular message since, due to the synchronous nature of the timing model, a message can be identified as not sent. In contrast, a fault is *malicious* if the faulty process can send any message to its neighbors, including false messages. A fault is *asymmetric* if a faulty process are not forced to send the same information to each of their neighbors. In contrast, in a *symmetric fault*, a faulty process sends the same (possibly wrong) message to all neighbors. In other words, faulty processes lie consistenly to all other processes.

As mentioned above, we assume that processes can only communicate by means of two-party messages, that is, we assume a fully connected topology for the processes. In other words, the neighbor set of a process contains all other $n-1$ processes.

One key assumption we make about the behavior of faulty process is that a faulty process always updates its local state correctly (just like a non-faulty process). The faulty behavior of a process is manifest only through the possibly faulty messages it sends to its neighbors. There are many different ways to motivate this assumption. One way is to view the processes as non-faulty and the communication channels (specifically, all outgoing channels from a faulty process) to be faulty. A second way is to note that this assumption makes no difference in case of permanent faults. It is relevant only for transient faults, and in a transient fault, a faulty process could become non-faulty and hence it could update its internal state correctly.

Rather than working with an arbitrary number n of processes out of which some f are faulty, we will work with concrete instances in this paper. Specifically, we will focus on $n = 4$ or $n = 3$ processes out of which $f = 1$ will be faulty.

Finally, in our faulty distributed system setting, we have to appropriately redefine our desired "agreeement state". Note that, in the scenario where a certain process is always faulty and never shares its correct local value, the other processes can not ever know the correct value of such faulty process and hence never reach agreement on the correct internal values of all processes. The notion of "interactive consistency" was, therefore, introduced to describe a final agreement configuration where all nonfaulty processes know the correct internal value of all (other) nonfaulty processes, and agree on the same (maybe wrong) value for the faulty processes. When faults are transient, the identity of the faulty process keeps changing. Therefore, we adapt the definition of interactive consistency so that it does not mention faulty and non-faulty processes and instead just talks about the number of processes that are in agreement (Definition 1).

Our main result is an algorithm for achieving interactive consistency in presence of non-malicious, transient, and asymmetric faults. Our algorithm achieves interactive consistency in three rounds, *and then preserves it forever thereafter*, and allows one (possibly different) process to be faulty in each round. If our algorithm is run for $f \geq 3$ rounds, we tolerate a total of f faults in f rounds. We overcome the impossibility result (which says that $f + 1$ rounds are needed for tolerating f faults) by introducing a *guaranteed delayed ack* assumption. Our algorithm was synthesized automatically. We also describe our synthesis approach in this paper, which is a generic approach for synthesis, and is particularly promising for synthesizing distributed algorithms.

1.1 Formalization and Notation

We assume there are n processes that are formally just elements of the set $\overline{n} = \{1, \ldots, n\}$. Each process i has a private Boolean value. Each process (say, Process i) is assumed to maintain a local *consistency vector* $\mathtt{cv}^{(i)} := (v_1, \ldots, v_n)$ where v_j is Process i's estimate of Process j's private value and belongs to the set $\mathit{Vals} := \{\mathtt{true}, \mathtt{false}, \mathtt{nil}\}$. For every i, the component v_i of the consistency vector of Process i is the private Boolean value for Process i. Processes do not change their own private value in their consistency vector, but do update their estimate of the private value of other processes in their local consistency vector.

A *message* is a tuple (s, v), where s is a finite string over the alphabet $\overline{n}$; that is, $s \in \overline{n}^*$, where $A^* = \cup_i A^i$, and $v \in \mathit{Vals}$ is a value. The length of the string s is at least 2. The tuple $(12, \mathtt{true})$ denotes the message sent by Process 2 to Process 1 informing Process 1 that Process 2's private value is $\mathtt{true}$. If the length of s is greater-than 2, then the meaning of the message (s, v) is defined inductively. The tuple $(12s, v)$ denotes the message that is forwarded by Process 2 to Process 1 informing Process 1 that Process 2 had received the message $(2s, v)$ in the previous round. For example, $(123, \mathtt{true})$ is the message forwarded by Process 2 to Process 1 which Process 2 had received from Process 1 in the previous round (containing the value $\mathtt{true}$). Note that if Process 1 receives $(123, \mathtt{true})$, then Process 1 receives the string 123 as well as the value $\mathtt{true}$.

We remark here about the implicit assumption being made above. We are assuming that nodes have an "identity" and processes know which process sent what message to it. So, for example, the scenario "Process 1 receives the messages $(12, \mathtt{true})$ and $(13, \mathtt{false})$" is different from the scenario "Process 1 receives the messages $(12, \mathtt{false})$ and $(13, \mathtt{true})$". If Process 1 could not distinguish between the different senders, then the two scenarios would look identical to Process 1: in both cases, Process 1 receives one $\mathtt{true}$ and one $\mathtt{false}$ message.

We assume that processes are *deterministic*. In each round, Process i receives messages (sent to it by the other processes in the previous round), and updates its local consistency vector using some deterministic function of its old consistency vector and the received messages. Furthermore, Process i also generates, and then sends, messages to other processes. As we will later see, on most occassions, these messages are just *forwarded* messages.

Definition 1 (Interactive Consistency). *A set of n processes, out of which at most f can be faulty in any given round, are said to have achieved interactive consistency if the local consistency vector of some $n - f$ processes are identical.*

The definition of interactive consistency given above is a generalization of the definition given by Pease, Shostak and Lamport [12] to the case when faults can be transient. Their definition used the identity of the faulty process, and makes sense when faults are permanent.

Interactive consistency is the *agreement* requirement. Distributed consensus algorithms additionally are required to satisfy *validity* and *termination* requirements. Validity is implicitly present in our formulation via the assumption that (a) processes are not allowed to change their private value, combined with the assumption that (b) Process i's consistency vector stores its private value at vector's i-th component.

Fault attribute	Characterization
Permanent	$\forall r : \texttt{fault}(r) = \texttt{fault}(1)$
Transient	Does not satisfy constraint for permanent fault
Benign	$\forall \texttt{msg}(\texttt{ij}, \texttt{v}) : v = \texttt{nil} \vee v = \texttt{cv}^{(j)}[j]$
Malicious	$\forall \texttt{msg}(\texttt{ij}, \texttt{v}) : v \in \mathit{Vals}$
Symmetric	$\forall \texttt{msg}(\texttt{ij}, \texttt{v}_1), (\texttt{kj}, \texttt{v}_2) : v_1 = v_2$
Asymmetric	Faulty node j need not satisfy above constraint

Table 1. Formal characterization of the fault attributes used in this paper.

Fault Attributes. Let $r \in \{1, 2, \ldots\}$ denote the round number and let $\texttt{fault}(r)$ denote the set of processes that are faulty in Round r. Table 1 contains the formal characterization of the different fault attributes used in this paper.

Problem Statement. We wish to find an algorithm for achieving interactive consistency in presence of transient, non-malicious, and asymmetric faults. By achieving we mean
(a) the system should reach an interactive consistent state after a finite number of rounds (termination property), and
(b) the interactive consistency property is preserved ad infinitum thereafter.

Part (b) of the requirement above is usually not included in classic distributed consensus. For permanent faults, it is irrelevant, but for transient faults, it is an important requirement. In linear temporal logic, the two parts together define an *eventually-always* (FG) property, but with the difference that the F operator is a bounded-F operator.

2 Non-transient, Malicious and Asymmetric Faults

Pease, Shostak, and Lamport presented in [12] an algorithm to achieve interactive consistency among n processes with a synchronous timing model and a permanent, malicious, and asymmetric fault model. Their algorithm is based on rounds of message exchanges and can withstand f faulty processes, as long as $n \geq 3f + 1$ holds. In this section we informally present Pease, Shostak, and Lamport's algorithm for the particular case when $n = 4$ and $f = 1$, and show that it does not work if we allow faults to be transient, even if we restrict them to be non-malicious.

In the particular case where $n = 4$ and $f = 1$, Pease, Shostak, and Lamport's algorithm achieves interactive consistency after two rounds of information exchange. In the first round, each non-faulty process sends its private value to every other process and, in the second round, the processes exchange the information obtained in the first round. Hence, for instance, process 1 receives messages $(12, v_2)$, $(13, v_3)$, $(14, v_4)$ in the first round, and messages $(123, v_{23})$, $(124, v_{24})$, $(132, v_{32})$, $(134, v_{34})$, $(142, v_{42})$, $(143, v_{43})$ in the second round. Then, each process i updates the jth component of its consistency vector, with $j \neq i$, according to the *three* received messages that report about j's value: if two of the values in these three messages coincide then i updates $\texttt{cv}^{(1)}[2]$ to that common value. Otherwise, i sets $\texttt{cv}^{(i)}[j]$ to $\texttt{nil}$. For example, process 1, would determine the value of $\texttt{cv}^{(1)}[2]$ using the values v_2, v_{42}, v_{32} of the received messages shown above.

To informally argue about the correctness of the algorithm, let us represent graphically, as a directed acyclic graph, the exchanges of messages that are relevant for Process 1 (left) and Process 3 (right) to determine the private value of 2.

$$\begin{tikzcd}
 & 1 \arrow[dr, "v_2"] & \\
2 \arrow[ur] \arrow[r] \arrow[dr] & 3 \arrow[r, "v_{32}"] & 1 \\
 & 4 \arrow[ur, "v_{42}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & 3 \arrow[dr, "u_2"] & \\
2 \arrow[ur] \arrow[r] \arrow[dr] & 1 \arrow[r, "u_{12}"] & 3 \\
 & 4 \arrow[ur, "u_{42}"'] &
\end{tikzcd}$$

The paths starting from a root node of the DAG represent messages. The path 231 in the left DAG represents the message $(132, v_{32})$ where Process 3 says to Process 1 in Round 2 that it received value v_{32} from Process 2 in Round 1. The path 211 in the left DAG denotes that, after Round 2, Process 1 has access to the value v_2 that Process 2 had sent to Process 1 in Round 1.

Assume that 1 and 3 are not faulty and let us first consider the case where 2 is faulty. Note that, in that case, $v_{42} = u_{42}$, $v_2 = u_{12}$, and $v_{32} = u_2$ hold, and hence 1 and 3 update $\texttt{cv}^{(1)}[2]$ and $\texttt{cv}^{(3)}[2]$ to the same value. Now consider the case where 2 is not faulty and thus 4 is the faulty node. In this case we have that $v_2 = u_{12} = v_{32} = u_2 = \texttt{cv}^{(2)}[2]$ holds, and both 1 and 3 update $\texttt{cv}^{(1)}[2]$ and $\texttt{cv}^{(3)}[2]$ to 2's private value. Note that, although we argued about processes 1 and 3 for clarity, we actually showed that every pair of non-faulty processes agree on a value for a faulty one and correcly infer the private value of a non-faulty one and hence the algorithm achieves interactive consistency.

We now show that Pease, Shostak, and Lamport's algorithm fails to achieve interactive consistency if faults are transient, even if the fault model is non-malicious. We give a counterexample using the same dag representation that we used above. However, in the transient case we must specify which process $i \in \{1, 2, 3, 4\}$ is faulty at each round. We denote such faulty process as $\bar{i}$. In our counterexample 2 is faulty in the first round and 1 is faulty in the second round. Note that, since the fault model is non-malicious, the private value of 2 must be `true` and, after the first two rounds, 3 and 4 will agree on the value of 2 to be `nil`, while 2 and 1 will agree on it to be `true`, which violates the interactive consistency property.

$$\begin{tikzcd}
 & \bar{1} \arrow[dr, "\texttt{true}"] & \\
\bar{2} \arrow[ur, "\texttt{true}"] \arrow[r, "\texttt{true}"] \arrow[dr, "\texttt{nil}"'] & 3 \arrow[r, "\texttt{true}"] & 1 \\
 & 4 \arrow[ur, "\texttt{nil}"'] &
\end{tikzcd}
\quad
\begin{tikzcd}
 & 4 \arrow[dr, "\texttt{nil}"] & \\
\bar{2} \arrow[ur, "\texttt{nil}"] \arrow[r, "\texttt{true}"] \arrow[dr, "\texttt{true}"'] & \bar{1} \arrow[r, "\texttt{nil}"] & 4 \\
 & 3 \arrow[ur, "\texttt{true}"'] &
\end{tikzcd}
\quad
\begin{tikzcd}
 & 3 \arrow[dr, "\texttt{true}"] & \\
\bar{2} \arrow[ur, "\texttt{true}"] \arrow[r, "\texttt{true}"] \arrow[dr, "\texttt{nil}"'] & \bar{1} \arrow[r, "\texttt{nil}"] & 3 \\
 & 4 \arrow[ur, "\texttt{nil}"'] &
\end{tikzcd}$$

The reader might wonder: if 2 is non-faulty in Round 2, then if we start a new instance of the Pease, Shostak and Lamport's algorithm in Round 2 (so that the new instance's Round 1 will happen along with Round 2 of the first instance in Step 2), then in Step 3 we would likely reach agreement. This is true, but in this case, there is another scenario in which agreement is not guaranteed after any fixed number of rounds. Consider the case when Process 2 is faulty in the first k steps, and in the $k+1$-th step, Process 1 becomes faulty (as shown above). In this case, the interactive consistency property holds in steps $2, 3, \ldots, k$, but is violated in Step $k+1$. Since k can be arbitrary, at no step can the processes know that they have achieved interactive consistency *from hereon*.

The above observation is not surprising: our formulation of the problem under the transient fault model can be viewed as there being f faults in f rounds (for $n = 4$ processes). It is known that tolerating f faults requires at least $f + 1$ rounds (so that existence of at least one fault-free round is guaranteed); see Theorem 6.33 in [10]. Our goal is to tolerate f faults in f rounds, but under an additional assumption. Note that Pease, Shostak and Lamport's algorithm tolerates 2 faults in 2 rounds, but *it assumes that the same process is faulty in both rounds*.

Our assumption, which we call the *guaranteed delayed ack* assumption, is as follows: whenever Process i sends a value (or a message) to Process j in Round r, then, in Round $r+2$, it knows the value (or message) Process j received from it in Round $r+1$. Intuitively, after 2 rounds, the sender gets an confirmation about whether its message was delivered or dropped. Formally, for every i, we define a function $\texttt{conf}_i$ that given a message path iji returns value v if (ji, v) was a message that was seen in the previous round:

$$\texttt{conf}_i(iji) = v \text{ if } (ji, v) \text{ was a message sent in previous round}$$

We assume that Process i has access to function $\texttt{conf}_i$ in Round 3 of our protocol.[1]

3 Algorithm for Interactive Consistency under non-malicious, asymmetric, transient faults

In this section we present an algorithm for interactive consistency in presence of permanent, malicious, and asymmetric faults for the case where $n = 4$ and $f = 1$. The algorithm was synthesized with the help of automated synthesis techniques described in the next section. In this section, we just describe the algorithm and present an informal proof for its correctness.

Similar to Pease, Shostak, and Lamport's algorithm, our algorithm is based on rounds of message exchanges. However, we use three rounds instead of two to reach a state that satisfies the interactive consistency property. We present our procedure as a nonterminating procedure that preserves interactive consistency in every step after the first three steps. Note that the preservation property is not trivial: since the fault model is transient, different processes can become faulty in different rounds and can potentially cause violation of interactive consistency.

[1] Note that the Pease, Shostak and Lamport's algorithm is not designed to benefit from such an assumption. Even if the assumption is made stronger and we let a faulty process know *in the next round* (that it was faulty in the previous round), the Pease, Shostak and Lamport algorithm can not use this fact since the faulty process does not participate in the message exchanges that are used to decide on it's local value.

Inputs: local value l_i for each $i \in \overline{n}$
Global: consistency vector $\texttt{cv}^{(i)}$; Initialized such that $\forall i$: $\texttt{cv}^{(i)}[i] = l_i \wedge \forall j \neq i : \texttt{cv}^{(i)}[j] = \texttt{nil}$
Output: consistency vector $\texttt{cv}^{(i)}$ that always satisfies the invariant
$\exists i \in \overline{n}$ s.t. $\texttt{cv}^{(j)}$'s are identical for all $j \neq i$.

```
IC_{4,1}:                // Describing a round for Process $i$
  $R = \texttt{receiveMessages}()$
  For $j \in \overline{n},\ \ j \neq i$   Do
    $D := \{v \mid (iiij, v) \in R\} \cup \{v \mid \exists x \neq i, x \neq j : (ixxj, v) \in R \vee (ijxj, v) \in R\}$
    If $\exists v \in D : v = \texttt{true}$ Then $\texttt{cv}^{(i)}[j] = \texttt{true}$
    ElseIf $\exists v \in D : v = \texttt{false}$ Then $\texttt{cv}^{(i)}[j] = \texttt{false}$
    Else $\texttt{cv}^{(i)}[j] = \texttt{nil}$
  $S_1 := \{(ji, \widetilde{l_i}) \mid j \in \overline{n}\}$
  $S_2 := \{(jix, \widetilde{v}) \mid (ix, v) \in R \wedge i, x \in \overline{n}\}$
  $S_3 := \{(jixy, \widetilde{v}) \mid (ixy, v) \in R \wedge y \neq i \wedge x, y, j \in \overline{n}\}$
  $S_{conf} = \{(jixi, \widetilde{v}) \mid \texttt{conf}_i(ixi) = v \wedge x, y, j \in \overline{n}\}$
  $\texttt{sendMessages}(S_1 \cup S_2 \cup S_3 \cup S_{conf})$
  where $\widetilde{v} = v$ if $i$ is not faulty and $\widetilde{v} \in \{v, \texttt{nil}\}$ if $i$ is faulty
```

Fig. 1. Algorithm for interactive consistency (n=4, f=1)

Our algorithm is presented in Figure 1. Our algorithm exchanges the same messages as the Pease, Shostak and Lamport algorithm in the first two rounds. The third round is introduced to add a redundant channel of communication thanks to the *guaranteed delayed ack* assumption introduced in the previous section.

Recall the notation about messages: the message (ji, v) represents that Process j receives value v from Process i in this round, and the messages $(jjji, v)$ and (jji, v) represent that Process j receives the value v from Process i two and one rounds back, respectively. Note that (ji, v) involves a message exchange, but (jji, v) does not involve any message exchanges and is just a convenient notation for describing information from one round back.

In the algorithm in Figure 1, at each step, every process first receives messages (containing information from up to three rounds back), then updates its consistency vector, and then sends messages. Every process sends information refering to one, two, and three steps back; that is, messages of the form (s, v), with s of lengths 2, 3, and 4.

The rule that Process i uses for updating its consistency vector is as follows: $\texttt{cv}^{(i)}[i]$ is left unchanged; and for all $j \neq i$, $\texttt{cv}^{(i)}[j]$ is set to a *non-nil* value v if either
(1) Process i receives a message $(iiij, v)$, or
(2) Process i receives a message $(ixxj, v)$ for some x different from i and j, or
(3) Process i receives a message $(ijxj, v)$ for some x different from i and j.

Apart from the messages of the form $(ijxj, v)$, all other messages have the usual meaning. This becomes clear from the code in Figure 1 that constructs the messages to be sent. Specifically, the value v in the message $(ijxj, v)$ is not equal to the value of the message path jxj, but it is equal to $\texttt{conf}_j(jxj)$. The notation $\widetilde{v}$ in Figure 1 is used to denote v if the process sending (or forwarding) value v is non-faulty, and it is non-deterministically picked from the set $\{v, \texttt{nil}\}$ if the process is faulty.

Example 1. Let us consider an example where Processes $1, 3, 4$ are trying to learn the local value l_2 of Process 2. First assume no process in faulty in any of the rounds. In this case, Process 2 sends l_2 to all three processes in Round 1; that is, in Round 0, we have

$$\forall i : (i2, l_2) \in S_1, \quad S_2 = S_3 = S_{\texttt{conf}} = \emptyset$$

After Round 1 (focusing only on value sent by Process 2), we have

$$\forall i, j : (ji2, l_2) \in S_2, \quad S_3 = S_{\texttt{conf}} = \emptyset$$

After Round 2 (focusing only on value sent by Process 2), we have

$$\forall i, k : \forall j \neq 2 : (kji2, l_2) \in S_3, \quad \forall i, k : (k2i2, l_2) \in S_{\texttt{conf}}$$

So, in Round 3, every Process i updates its consistency vector to have value l_2 in $\texttt{cv}^{(i)}[2]$ – since, for example, $(iii2, l_2)$ is received by every Process i.

As a more interesting example, consider the scenario where Process 2 is faulty in Round 0 and sends its local value l_2 to only Process 3 (and $\texttt{nil}$'s to others); and moreover, Process 3 becomes faulty in Round 1 and does not forward the correct message it received from Process 2 to others. In Round 2, if Process 2 is non-faulty, then it will send the message $(i232, l_2)$ to all processes i (these messages will belong to the set $S_{\texttt{conf}}$ is Round 2). Consequently, in Round 3, all processes i will update $\texttt{cv}^{(i)}[2]$ to l_2. But, what if Process 2 becomes faulty in Round 2? In that case, Process 3 is not faulty, and hence it forwards correctly; that is, $(i332, l_2)$ is in the set S_3 of sent messages in Round 2. Consequently, in Round 3, all processes i will again update $\texttt{cv}^{(i)}[2]$ to l_2.

We will argue informally about the correctness of the algorithm of Figure 1 using again the DAG representation introduced in the previous section. It is easy to see that if all processes execute the algorithm of Figure 1 and a faulty process, say 2, does not reveal its local value to any other process for the first $k > 0$ steps, i.e. sends messages $(j2, \texttt{nil})$ to every process $j \neq 2$, then $\texttt{cv}^{(j)}[2] = \texttt{nil}$ holds in steps k, $k+1$, and $k+2$, for every process $j \neq 2$. Hence, note that it suffices to get convinced that, if in step $k+1$ Process 2 sends its local value to some process, say Process 3, then $\texttt{cv}^{(j)}[2] = l_2$ will hold in step $k + 4$, for every process j. Consider the following two DAGs representing the exchanges of messages that are relevant for processes 3 and 1 to decide about the value of Process 2 and assume that 2 sends $(32, l_2)$.

$$\begin{tikzcd}
 & 1 \arrow[r] & 1 \arrow[dr] & \\
2 \arrow[ur] \arrow[dr, "l_2"] \arrow[dd, dash, dashed] & & 2 \arrow[r, "v_1"] & 1 \\
 & 3 \arrow[r] \arrow[ur, dotted] & 3 \arrow[ur, "v_2"'] & \\
\text{Step } k+1 & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & 3 \arrow[r, "l_2"] & 3 \arrow[dr, "l_2"] & \\
2 \arrow[ur, "l_2"] \arrow[dr] \arrow[dd, dash, dashed] & & 2 \arrow[r] & 3 \\
 & 1 \arrow[r] \arrow[ur, dotted] & 1 \arrow[ur] & \\
\text{Step } k+1 & & &
\end{tikzcd}$$

Note that, due to the *guaranteed delayed ack* assumption, either $v_1 = l_2$ or 2 is faulty in round $k+3$ and $v_2 = l_2$. In any case, Process 1 updates $\mathtt{cv}^{(1)}[2]$ to l_2 in round $k+4$. Moreover, note that Process 4 also updates $\mathtt{cv}^{(4)}[2]$ to l_2 in the same round by a symmetric argument. Finally, Process 3 updates $\mathtt{cv}^{(3)}[2]$ to l_2 in round $k+4$ because it receives the message $(3332, l_2)$.

4 A General Synthesis Approach for FG Properties

In this section, we will outline the synthesis approach we used to arrive at the (variants of the) algorithms presented in the previous section.

All modern synthesis tools work by enumerating the space of possible solutions and checking if one of these solutions satisfies the requirement. Checking if a synthesized solution satisfies a requirement is a formal verification problem. Broadly speaking, synthesis is performed as a loop over the formal verification tool. Our approach to synthesis is simpler and can be viewed as a generalization of the idea of bounded model checking to synthesis. Just as bounded model checking turns a verification problem into a *existential* contraint that encodes a *weaker* version of the verification problem, we turn synthesis into a *forall-exists* constraint that encodes a *weaker* version of the synthesis problem.

The key step that makes automated synthesis effective is the step that defines the weaker version. A simpler version of the synthesis problem is obtained by
(a) restricting the universe of possible algorithms that will be searched and
(b) replacing the verification step by an approximate step.

In particular, we make the search space of possible solutions finite. In the context of synthesis of distributed consensus, this is achieved by first fixing the number of processes (to a small number such as 4). Then, we fix the *type of messages* that are exchanged in different rounds: in our case, we fix the component s of the messages (s, v) that are exchanged in a round and we even fix the computation of the value v of the message in most cases. We then just need to synthesize the *deterministic function* that is used to update the local consistency vector and generate the value v for the messages (if they are not already fixed) for the next round. The domain and range of the function is finite. Hence, there are only finitely many, but a huge number nonetheless, of such functions.

We also make the verification problem simple – instead of performing full verification, we just perform bounded model checking (checking up to some fixed bound). In the context of synthesis of distributed consensus, we fix the number of rounds (say, to 3) and then search for algorithms that achieve agreement in exactly 3 rounds. A bounded model checker is used for this purpose. Thus, we synthesize for agreement and termination requirements, while validity is built-in in the formulation of the synthesis problem. We ignore stability requirement while performing synthesis.

We now present a more formal description of the synthesis approach described above. Let $\mathtt{x}$ denote all the state variables of the distributed system. In our case, $\mathtt{x}$ contains the consistency vector of all processes. Let $\mathtt{y}$ denote the variables representing the *values* of the messages that are exchanged in any round. Let $\phi(\mathtt{x})$ be the formula that encodes the property that $\mathtt{x}$ is the desired final state (that is, $\mathtt{x}$ is an interactive-consistent state.) The following *verification constraint*, generated by a bounded model checker, says that there is a sequence of 4 states of the system that follows the consensus algorithm, but does not end in an agreement state (it is the negation of the what we want to prove):

$$\begin{aligned}&\exists \mathtt{x}_0, \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{y}_0, \mathtt{y}_1, \mathtt{y}_2 : \\ &\quad I(\mathtt{x}_0) \wedge T(\mathtt{x}_0, \mathtt{y}_0, \mathtt{x}_1) \wedge T(\mathtt{x}_1, \mathtt{y}_1, \mathtt{x}_2) \wedge T(\mathtt{x}_2, \mathtt{y}_2, \mathtt{x}_3) \wedge \neg\phi(\mathtt{x}_3)\end{aligned} \tag{1}$$

Here $I(\mathtt{x})$ is a predicate that is true if $\mathtt{x}$ is a valid initial state and $T(\mathtt{x}, \mathtt{y}, \mathtt{x}')$ is a predicate that is true if $\mathtt{y}$ are the messages that would be generated in state $\mathtt{x}$ and $\mathtt{x}'$ would be the next state generated from current state $\mathtt{x}$ and these messages.

For a given deterministic consensus algorithm, the predicate $T(\mathtt{x}, \mathtt{y}, \mathtt{x}')$ is a function from $(\mathtt{x}, \mathtt{y})$ to $\mathtt{x}'$, but it is not a function from $\mathtt{x}$ to $(\mathtt{y}, \mathtt{x}')$ since different manifestation of the faults can cause different messages to be generated from the same state $\mathtt{x}$. So, the verification constraint says that consensus algorithm does not achieve agreement in 3 steps for some choice of initial state and fault behavior.

When we synthesize the consensus algorithm, the predicate T is not fully known. It is, in fact, parameterized by some additional *synthesis variables* $\mathtt{z}$ such that the new relation $T(\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{x}')$ is a function from $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ to $\mathtt{x}'$. So, the *bounded synthesis constraint* is defined as

$$\begin{aligned}&\forall \mathtt{z} : \exists \mathtt{x}_0, \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{y}_0, \mathtt{y}_1, \mathtt{y}_2 : \\ &\quad I(\mathtt{x}_0) \wedge T(\mathtt{x}_0, \mathtt{y}_0, \mathtt{z}, \mathtt{x}_1) \wedge T(\mathtt{x}_1, \mathtt{y}_1, \mathtt{z}, \mathtt{x}_2) \wedge T(\mathtt{x}_2, \mathtt{y}_2, \mathtt{z}, \mathtt{x}_3) \wedge \neg\phi(\mathtt{x}_3)\end{aligned} \tag{2}$$

The synthesis constraint says that forall choices of the synthesis variables $\mathtt{z}$, the resulting consensus algorithm does not achieve agreement in 3 steps (for some initial state and fault behavior). If we do not want to fix *a priori* the function that determines what message values $\mathtt{y}$ are generated in a state $\mathtt{x}$ (for a fixed choice of faulty nodes), then we can also synthesize that function by including additional parameters in $\mathtt{z}$ that are used to define that function.

In our case, the domain of all variables in Formula 2 have finite cardinality. Hence, the formula can be written as a quantified ($\forall\exists$) Boolean formula (QBF). Bounded model checkers (such as the SAL bounded model checker we used) already generate a Boolean satisfiability (SAT) formula for the verification constraint (Formula 1).

Our synthesis approach implementation consists of a script that glues together different tools as follows:

1. We model the consensus algorithm in SAL [13, 2]. The model includes synthesis variables $\mathtt{z}$ to define the transition relation.
2. We use the SAL bounded model checker to generate the SAT formula for the verification constraint (Formula 1). The SAT formula implicitly existentially quantifies all variables, including the synthesis variables $\mathtt{z}$.
3. We modify the SAT formula and convert it into a QBF formula by universally quantifying the synthesis variables. (This step is tricky because it needs the mapping from the original SAL variables to the Boolean SAT variables.)
4. We use off-the-shelf QBF solvers (and QBF preprocessors) to check satisfiability of the $\forall\exists$ formula.

5. If the QBF solver returns `Unsat`, then the synthesis is declared *successful*, and if the QBF solver returns `Sat`, then the synthesis process is *unsuccessful*.
6. If synthesis is successful, the QBF solver outputs a valuation for the synthesis variables `z`, which is used to obtain a concrete consensus algorithm.
7. The synthesized algorithm is formally verified: the property that "after 3 steps, the property ϕ is always true" is verified using k-induction.

In many cases when synthesis was successful, the valuation for the synthesis variables `z` returned by the QBF solver was not easy to describe; that is, the resulting update function did not have a concise description. This happens because QBF solver would instantiate "don't care" variables arbitrarily. These algorithms were not suitable for describing in this paper. Hence, in such cases, we used our intuition to modify the synthesized function so that it had a concise description, and then verified the modified algorithm and presented it here.

5 Synthesis Problem Formulation and Experimental Results

As mentioned in the previous section, we restricted the space of possible algorithms to be searched by the synthesis tool to a finite set. The major restriction, apart from fixing n to be 4, is only considering algorithms based on rounds of message exchanges, in contrast with other kinds of distributed algorithms based in other schemas such as, for example, a fixed process acting as the leader.

Hence, while the dynamics of the messages exchanges are fixed, the task of the synthesis tool is to decide how to update the consistency vector depending on the messages received at each step. Note that this corresponds to the For-Do loop of Figure 1. More specifically, the goal of our tool was to synthetize a deterministic function $f^k_{i,j}$, which corresponds to the algorithm executed by Process i to update $\texttt{cv}^{(i)}[j]$ given the messages received in the last k rounds reporting about the value of j. Hence, each function $f^k_{i,j}$ in this family of functions parameterized by k has the following signature:

$$f^k_{i,j} : \{i, j, k, l\}^{k-1} \times \Sigma \mapsto \Sigma$$

where $\Sigma = \{\texttt{true}, \texttt{false}, \texttt{nil}\}$ and k is the number of rounds of message exchanges to be considered by the synthesized interactive consensus algorithm. The synthesis of $f^k_{i,j}$ is subject to the constraint that the resulting algorithm achieves interactive consistency. Such constraint, as well as the fault model were very naturally encoded as an LTL property and part of a SAL model, as explained in the previous section. For example, assuming $k = 2$, as in the Pease, Shostak and Lamport's algorithm, our synthesis problem consists on deciding how Process i must update $\texttt{cv}^{(i)}[j]$ given messages $(iij, v_1), (ijj, v_2), (ikj, v_3), (ilj, v_4)$, for every possible value of $v_1, v_2, v_3, v_4 \in \{\texttt{true}, \texttt{false}, \texttt{nil}\}$ and assuming that i, j, k, l are pairwise disjoint. Using this approach we could easily synthesize Pease, Shostak and Lamport's algorithm for $n = 4$. Similarly, we could prove that there is no interactive consistency algorithm for the non-malicious, asymmetric, and transient case that is based in two rounds of message exchange, even if we assume guaranteed delayed ack.

Note that the domain of $f^k_{i,j}$, although finite, has exponential size with respect to k. In fact, in the case where $k = 3$, it has size 48, which corresponds to a search space for the synthesis process of size 3^{48}. However, to speed up the synthesis process, we reduced the size of the image of $f^3_{i,j}$ by
(a) not considering combinations of messages that are not possible due to the characteristics of the fault model (note that Process i cannot receive messages reporting j's local value to be both `true` and `false` since faults are non-malicious), and
(b) not considering some messages that intuitively seemed to be unnecesary for the algorithm (for example, the message $(ijij, v)$, is clearly useless in our setting).

In fact, although the version of the algorithm presented in Section 3 uses only the five messages corresponding to the paths $iiij, ikkj, illj, ijkj, ijlj$, our first synthetized version used also $iikj$ and $iilj$, which intuitively correspond to Processes k and l, respectively, telling Process i in the second round the value that they got from j in the first round. Later on we realized that these messages were indeed unnecesary, and we could synthetize a solution that ignores them.

With respect to impossibility results, we could use our tool to prove some particular cases of Theorem 6.33 in [10] that arise from fixing a particular message exchange dynamics such as variants the idea of running two overlapping instances of Pease, Shostak and Lamport's algorithm as commented in Section 2.

In all our experiments we used the QBF solver DepQbf [9] and the QBF preprocessor Bloqqer [1] and obtained the results in the order of minutes. Moreover, all our synthetized algorithms were verifiable using k-induction, which was proven much more effective than symbolic model checking. The sal model used to obtain our main result, as well as the corresponding QBF formula are available from www.csl.sri.com/users/tiwari.

6 Discussion

Distributed algorithms are difficult to design because of the enourmous number of scenarios generated due to the faults. We found our synthesis tool to be extremely useful in the process of identifying existence of algorithms of a certain form that achieve a certain goal.

We were able to synthesize the original Pease, Shostak and Lamport's algorithm for $n = 4$ processes and $f = 1$ fault, where the fault was permanent (across the two rounds) and asymmetric – irrespective of whether the fault was malicious or not. The synthesis tool declared "synthesis unsuccessful" when we changed the fault model to transient – both when the fault was malicious and when it was not malicious. We also tried to perform synthesis under slightly different fault models. For instance, when Process 1 receives a message $(123, \texttt{nil})$, then we allowed Process 1 to know if 2 was faulty in the previous round, or if 3 was faulty in the round before (akin to *manifest* faults). However, synthesis failed in most of such minor variants on the fault model.

Generalizing to $n > 4$. Our synthesized algorithm, presented in Section3, is easy to generalize for larger values of n, but keeping the assumption that in each round at most one process is faulty. In fact, for any value of n larger than one, exactly the same algorithm generalizes and three rounds suffice to reach interactive consistency.

The case when more than one process is faulty in every round is not yet known to us. The synthesized algorithm does not naturally generalize to a working algorithm for this case.

Message Complexity. In our description of the algorithm, we assumed that arbitrary messages of the form $(ijkl, v)$ are exchanged. However, Process i uses only *five* messages in the end to decide on a value for Process j: the values corresponding to the message paths $iiij, ikkj, illj, ijkj, ijlj$: these involve only a total of $1+2+2+3+3 = 11$ messages (across all three rounds) for each pair i, j of nodes. We are counting every use of the $\texttt{conf}_j$ function as a message exchange.

Tolerating Malicious Faults with Authentication. The algorithm that works for non-malicious faults can also work for malicious faults, if we assume the processes authenticate their communications using digital signatures. Even though a faulty process is now allowed to send arbitrary messages, the receiver can check if a non-nil value was really sent by the originator and discard it (treat it as a nil value) if the check fails. This forces the faulty process to only possibly behave like a non-malicious faulty process.

Extensions. The results described in this paper are just a first step in application of synthesis technology for discovering fault-tolerant distributed algorithms. There are plenty of avenues to explore for future work. First, there are asymmetric architectures that can provide same level of fault tolerance with less hardware resource. For example, the Draper Laboratory's Fault Tolerant Processor (FTP) [6, 8] is an asymmetrical design that uses *interstages* to relay messages from a process to it's neighbors. Second, one can also consider hybrid fault models [8] in distributed consensus. Finally, one can also consider problems in distributed algorithms for automated synthesis that have requirements other than the consensus property.

Synthesis for Fault-Tolerance. Automated synthesis was first considered in the context of synthesizing from LTL specification [11]. Later, Kulkarni et al. [5] started with a fault-intolerant distributed algorithm and showed how to automatically transform it into a fault-tolerant program. The technique was based on refining the given program by removing states and transitions that lead to violation of agreement in presence of faults. Our formulation of the synthesis problem is inspired by recent work on Sketching [14, 15, 4] where the starting point is an incomplete sketch that is filled in by automated tools; in particular, by solvers for $\exists\forall$ formulas [4]. The use of sophisticated constraint solvers (SMT solvers, SAT and QBF solvers) allows our approach to discover completely unexpected algorithms from a huge search space.

7 Conclusion

We used automated synthesis to discover an algorithm for achieving interactive consistency in the presence of transient, non-malicious and asymmetric faults. Our algorithm can be seen as filling a known gap in the literature. One the one hand, it is known that there is no f round algorithm that achieves agreement in the presence of f non-malicious, asymmetric and transient faults. On the other hand, it is known that there is one such algorithm that achieves agreement in $f + 1$ rounds. Our algorithm achieves agreement in f rounds, but uses an extra assumption that we have called the guaranteed delayed ack assumption. The assumption allows a sender to know the value the receiver received from it, but only after an extra intermediate round of message exchanges.

Our synthesis approach for discovering distributed algorithms is based on bounded model checking and quantified boolean formula (QBF) solving, and has been an indispensable tool in our effort to obtain the above positive result, and also for showing the non-existence of a consensus algorithm for various other cases.

Acknowledgments.

We are greatly thankful to Patrick Lincoln and John Rushby for providing useful guidance throughout this work.

References Cited

1. A. Biere, F. Lonsing, and M. Seidl. Blocked clause elimination for QBF. In *CADE*, pages 101–115, 2011.
2. L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
3. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.
4. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. PLDI*, 2011.
5. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. In *20th Symp. on Reliable Distributed Systems, SRDS*, 2001.
6. J. H. Lala. A Byzantine resilient fault tolerant computer for nuclear power applications. In *Fault tolerant computing symposium*, pages 338–343, 1986.
7. L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
8. P. Lincoln and J. Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *Proc. 9th Conf. on Computer Assurance, COMPASS*, 1994.
9. F. Lonsing and A. Biere. DepQBF: A Dependency-Aware QBF Solver. *JSAT*, 7(2-3):71–76, 2010.
10. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
11. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 6:68–93, 1984.
12. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *JACM*, 27(2):228–234, 1980.
13. The SAL intermediate language, 2003. Computer Science Laboratory, SRI International, Menlo Park, CA. `http://sal.csl.sri.com/`.
14. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.
15. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.