Patterns of Communication in Consensus Protocols*

Cynthia Dwork†
Dale Skeen‡

TR 84-611
June 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

*An abbreviated version of this paper is to appear at the "Symposium on Principles of Distributed Computing," Vancouver, Canada, August 1984.
†Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139.
‡Department of Computer Science, Cornell University, Ithaca, New York, 14853.

Patterns of Communication in Consensus Protocols

Cynthia Dwork

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Dale Skeen

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

This paper presents a taxonomy of consensus problems, based on their safeness and liveness properties, and then explores the relationships among the different problems in the taxonomy. Each problem is characterized by the communication patterns of protocols solving it. This then becomes the basis for a new notion of reducibility between problems. Formally, problem P_1 reduces to problem P_2 whenever each set of communication patterns of a protocol for P_2 is the set of communication patterns of a protocol for P_1. This means intuitively that any protocol for P_2 can solve P_1 by relabeling local states and padding messages. Consequently, the message complexity (measured in number of messages) of P_1 is not greater than the message complexity of P_2. Our method of characterizing and comparing problems is the principal contribution of this paper.

1. Introduction

The ability of separated processors to reach consensus is a fundamental problem in distributed computation and has been studied extensively in the literature. (See Fischer [F] for a survey. Also see [DFFLS], [DRS], [GPD], [L83], [LPS], [PSL] for examples.) Generally, each processor begins with a binary value in its input register. At some point in the computation correct processors must irreversibly decide on a binary value. No two correct processors may decide differently. The details of the relationship of the initial values to the decision vary according to the particular version of the problem. Additional variations have been obtained by (1) varying the failure environment; (2) varying the assumptions on synchrony ([FLP, DDS, DLS]); (3) varying the notion of an atomic step ([DDS]); and (4) varying the range of acceptable decision values ([DLPSW]).

In practice consensus problems arise in numerous guises. The simplest of these is the *reliable broadcast* problem ([SGS]), better known as the *Byzantine Generals* problem ([PSL]). Other settings include transaction commitment systems ([DS], [Gr], [S82]), replicated file systems ([Gi]), resource allocation, and interpretation of sensor or other instrumentation readings ([W]).

In any fixed model (level of synchrony, type of failure, choice of atomic step, etc.) consensus problems seem to differ from one another in three principal aspects, and one contribution of this paper is a *taxonomy* for consensus problems corresponding to these parameters. The first parameter is the set of decision rules, i.e., conditions under which a processor can or must decide on a given value. For example, in the strong unanimity problem (see [F]) if all initial values are the same value, say v, then the decision must be v. The second parameter is the *consistency* constraint. In the reliable broadcast problem, only *nonfaulty* processors must agree on a value, while in the distributed commitment problem all processors that ever decide (including those that subsequently fail) must decide on the same value. The third parameter is the *termination*, or liveness, constraint. A frequently used termination requirement is simply that every nonfaulty processor eventually decide.

The goal of this paper is to unify work on the different forms of consensus problems by exploring the relationships among the different problems. To do this we define a new notion of reducibility. We first define for any protocol P a partial ordering on the message-sending steps of an execution of P (cf. Lamport [L78]). Intuitively, the sending of message m_1 precedes the sending of m_2 if and only if the contents of m_1 may be known to the sender of m_2 when m_2 is sent. We call this partial ordering the *communication pattern* of the execution.

For any protocol Q, let the *scheme* of Q denote the set of all communication patterns of failure-free executions of Q. A problem may be characterized by the set of schemes of protocols for the problem. We say P_1 *reduces to* P_2, written $P_1 \leq P_2$, if and only if the set of schemes for P_1 contains the set of schemes for P_2.

Intuitively, if P_1 reduces to P_2, then any protocol for P_2 can solve P_1 by relabeling local states and padding messages. Consequently, the message complexity (measured in number of messages) of P_1 is not greater than the message complexity of P_2. Our method of characterizing and comparing problems is the principal contribution of this paper. Given our taxonomy we use this notion of reducibility to examine the relationships among six practical problems with varying safeness and liveness properties.

2. A Taxonomy of Consensus Problems

In this section we briefly describe some possible choices for the three parameters mentioned in the introduction: decision rules, consistency constraints, and termination conditions. We assume a completely asynchronous model with fail-stop processors, meaning that processors fail by halting and that failures are detectable. Our model of computation is specified more fully in Section 3.

The most frequently used decision rule is the *Broadcast Rule*: decide v only if the initial value of a distinguished processor is v.[1] This is the decision rule of the Byzantine Generals problem. This rule, however, is inappropriate for problems such as transaction commitment, where input values of all the processors influence the decision. A common decision rule for these problems is ***unanimity***: decide 1 (commit) only if every processor's initial value is 1, and decide 0 (abort) only if some processor begins with value 0 or a failure occurs. Note that unanimity is meaningless in the presence of Byzantine failures, where processors can lie about their initial values.

There are obvious generalizations of the above rules, such as ***threshold-k***: decide 1 only if at least k processors have initial value 1; or ***set***(S,v): decide v only if all processors in set S have initial value v.

We identify two important consistency constraints. In ***interactive consistency*** (IC) no two operational processors may simultaneously occupy different decision states. In ***total consistency*** (TC) no two processors ever decide on different values. Notice that these constraints differ in their treatment of faulty, decided processors: in total consistency, any decision must be consistent with a decision made by another processor, even if that processor has subsequently failed. Total consistency is meaningless in a model allowing failed processors to make incorrect decisions. Total consistency is usually required when a decided processor could initiate an irreversible action, such as dispensing money.

We identify three increasingly strong types of termination. The weakest termination constraint considered here is ***weak termination*** (WT), which requires only that every nonfaulty processor decide within a bounded number of steps. Weak termination says nothing about when a processor can halt or even when it can forget about the particular execution of the protocol or about its decision. In fact, it admits solutions that never halt, even in failure-free executions. (Such protocols terminate, in essence, by deadlocking, with each processor listening for messages from its cohorts.)

Our two stronger termination conditions are intended for environments in which processors are repeatedly executing consensus protocols. Processors may even be executing several protocols at a time. In this situation we may imagine that all messages are tagged with a unique protocol identifier. If the set of possible decision values is large, it may be desirable to allow a processor to forget its decision for a given instance of a protocol, while remembering that a decision was made. We call this ***strong termination***. Figuratively, the processor places a check next to a record of the protocol identifier, indicating that a decision has been made but keeping no record of the processing involved. The resulting state is an ***amnesic*** state. In order to avoid talking about history, we will refer to a processor as being either in an ***amnesic commit*** or an ***amnesic abort state***, although there is really only one amnesic state. An amnesic processor may continue to send and receive messages. It may even be reminded of its decision by the other processors.

[1] This is the strong variant; the weak variant allows a default decision if the distinguished processor is faulty.

Another possibility is that we wish to allow a processor to complete its role in an execution of a protocol, in the sense that it need no longer send or receive messages relative to the given execution. We call this ***halting termination.*** Of course, a halted processor may fail, and its failure is detectable.

All possible combinations of the above rules and constraints have applications. For the Byzantine Generals problem, the combination broadcast rule, interactive consistency, and halting is normally assumed. However, weak termination, instead of halting, in used in the reliable broadcast protocols of [SGS] in order to reduce costs. For the transaction commitment problem, unanimity and total consistency are assumed, together with either weak termination ([S82]) or strong termination ([ML]).

3. Definitions and the Model of Computation

Our formal model of computation is based on the models of [FLP, DDS]. The processors are modeled as infinite-state machines with state set Z. At each of its ***steps,*** a processor may receive or send a message, but not both. In a receiving step it may change states according to its previous state and the contents of the message received. In a sending step it may send at most one message and change states. A third kind of step, a ***failure*** step, is discussed below.

A ***consensus protocol*** is a set of N processors, $P = \{p_0, p_1, \ldots, p_{N-1}\}$. As part of its state, each processor p_i has a set UP_i, initially containing all N processors. As p_i learns of failures it deletes these failed processors from UP_i. Each processor p_i also has an initial bit, $input_i$. There are two special ***initial states*** z_0 and z_1. For $v \in \{0,1\}$, a processor is started in state z_v if its initial bit is v. Each nonfaulty processor then follows a protocol involving the receipt and sending of messages. The messages are drawn from an infinite set M. Each processor has a buffer for holding the messages that have been sent to it but not yet received. The buffer is modeled as an unordered set of messages. The collection of buffers supports two operations:

$Send(p,m)$: places message m in p's buffer;

$Receive(p)$: delays p until a message is delivered, and deletes this message from p's buffer.

The message system is asynchronous, but it is also faultless and fair. A processor may suffer an arbitrary delay when executing a Receive operation, but if its buffer is nonempty, the delay is finite. In addition, in selecting a message to deliver to a processor, it will not discriminate against a given message infinitely often.

Each processor p is specified by a ***state transition function*** δ_p and a ***sending function*** β_p where

$$\delta_p: Z \times M \cup \{\emptyset\} \rightarrow Z$$
$$\beta_p: Z \rightarrow \{\emptyset\} \cup P-\{p\} \times M \cup \{(q, failed(p)) \mid q \in P-\{p\}\}.$$

The pair (q,m) in the range of β_p means that p sends message m to processor q. For technical reasons, p is not allowed to send a message to itself. In a normal (non-failure) step, a processor can send at most one message. In a failure step, a processor sends failure notices to all other processors. (This allows the other processors to detect the failure.)

We assume that Z is partitioned into three disjoint sets Z_R (the operational *receiving* *states*), Z_S (the operational *sending states*), and Z_F (the *failed states*). No normal messages are sent when in a receiving state (formally, if $z \in Z_R$ then $\beta_p(z) = \emptyset$). No messages are received when in a sending state. We also assume that Z contains two disjoint sets of *decision states* Y_0 and Y_1, such that if a processor enters a state in Y_v, $v \in \{0,1\}$, then it must remain in states in Y_v. (In the case of strong termination, processors are allowed to move from a decision state into an amnesic state.)

A *configuration* C consists of

(a) N states $state(p_i, C) \in Z$ for $1 \le i \le N$, specifying the current state of each processor, and

(b) N sets $buff(p_i, C) \in 2^M$ for $1 \le i \le N$, specifying the current contents of each buffer.

Initially, each state is either z_0 or z_1 as described above, each buffer is empty and each UP set contains all N processors.

An *event* is a pair (p,μ) where $p \in P$ and $\mu \in M \cup \{f, \emptyset\}$. If $\mu \notin \{f, \emptyset\}$ the event (p,μ) may be thought of as the receipt of message μ by processor p. Think of (p,f) as the event of p's failure. We now define conditions under which an event can be *applied* to a configuration to yield a new configuration.

(1) If $state(p,C) \in Z_S$, then (p,μ) is *applicable* to C only if $\mu = \emptyset$ or $\mu = f$.

(2) If $state(p,C) \in Z_R$, then (p,μ) is *applicable* to C if $\mu \in buff(p,C)$ or $\mu = f$.

If the event $e = (p,\mu)$ is applicable to C and e is not a failure transition, then the next configuration $e(C)$ is obtained as follows:

(a) p changes its state from $z = state(p,C)$ to $\delta_p(z,\mu)$ and the states of the other processors do not change,

(b) for all $(q,m) \in \beta_p(z)$, m is added to $buff(q,C)$,

(c) if $z \in Z_R$ and $\mu \ne f$ then μ is deleted from $buff(p,C)$.

A failure transition is modeled as two steps. We let $Z_F = \{z_a, z_b\}$. When a processor fails it first enters z_a, from which it broadcasts a failure notice. It then moves to state z_b. Formally,

(a) for all $z \notin Z_F$, $\delta_p(z,f) = z_a$,

(b) $\beta_p(z_a) = \{(q, failed(p)) \mid q \in P - \{p\}\}$,

(c) $\delta_p(z_a, \emptyset) = z_b$,

(d) for all μ, $\delta_p(z_b, \mu) = z_b$,

(e) $\beta_p(z_b) = \emptyset$.

Rules (d) and (e) ensure that once a processor has failed it cannot send messages or restart at a later time.

A *schedule* is a finite or infinite sequence of events. A schedule $\sigma = \sigma_1\sigma_2 \cdots$ is *applicable* to a configuration C if the events of σ can be applied in turn starting from C, i.e., σ_1 is applicable to C, σ_2 is applicable to $\sigma_1(C)$, etc. If σ is finite, $\sigma(C)$ denotes the resulting configuration, which is said to be *reachable* from C. A configuration reachable from some initial configuration is said to be *accessible*. Similarly, a local state s_p is accessible only if there exists a reachable configuration C and processor p such that $state(p,C) = s_p$. Henceforth, all configurations and states mentioned are assumed to be accessible.

A schedule together with the associated sequence of configurations is called a ***run***. A processor is ***nonfaulty*** in a run if it never occupies a failed state during the run. A run is a ***deciding run*** if every nonfaulty processor enters a decision state. An ***execution*** is a (possibly infinite) run from an initial configuration.

A processor's "knowledge" about the states of its cohorts is captured by the ***concurrency set*** of its state. The concurrency set of state s, denoted $C(s)$, is the set of states t such that s and t occur in the same configuration.

For an execution I of a given protocol, we wish to define a partial ordering ($<_I$) on the messages sent during the execution. The ordering is based on Lamport's "happens before" relation. Intuitively, $m <_I m'$ if the contents of message m could have influenced the contents of message m'. Since we will be interested in the ordering among messages, but not in their contents, we assume in the following definitions that a message is represented by a triple (p, q, k), meaning that the message was the k^{th} message sent from p to q.

Formally, the ordering $<_I$ is the smallest irreflexive, transitive relation satisfying:

(1) $m_1 <_I m_2$ if m_1 and m_2 have the same sender and m_1 is sent in real time before m_2 is sent;

(2) $m_1 <_I m_2$ if the recipient of m_1 is the sender of m_2 and m_1 is received before m_2 is sent.

The relation $<_I$ with messages represented as triples is called the ***communication pattern*** of I. The set of communication patterns of all failure-free executions of a protocol P is called the ***scheme*** of P.

As mentioned in the introduction, we characterize a problem by the set of sets of communication patterns (i.e., the set of ***schemes***) of protocols solving the problem. Let Q be a protocol for P_2. If P_1 ***reduces*** to P_2 ($P_1 \leq P_2$), then the scheme of Q is the scheme of some protocol for P_1. This says intuitively that Q is a protocol for P_1 up to a renaming of states and padding of messages. The $\leq$ relation is transitive. If $P_1 \leq P_2$ but the converse is false we write $P_1 < P_2$. Finally, if neither problem reduces to the other we say they are ***incomparable.***

We conclude this section by identifying a set of states that complicate reasoning about protocols. A processor only enters a state in this set if it knows its message buffer is not empty. This can happen if messages are delivered out of order. (If processors could send messages to themselves it could happen all the time.) We denote this set by $\overline{E}$ and its complement by E (for ***empty buffer***). A processor in an $\overline{E}$ state cannot be forced to make a decision: it can safely procrastinate until an impending message is delivered. A protocol P with $\overline{E}$ states can easily be transformed into a special ***total-communication*** protocol P' with no $\overline{E}$ states and whose communication patterns are a subset of the communication patterns of P. (Total-communication protocols are defined below, and the transformation is described.) If in the absence of failures the decision reached in P is a function of the inputs alone (and not, for example, of the order in which messages happen to be delivered in a particular execution of the protocol), then, in the absence of failures, P and P' compute the same function of the inputs. The unanimity decision rule enjoys this property.

A protocol is a *total-communication* protocol if whenever a processor sends a nonfailure message m_2 in an execution I, it appends a copy of every message m_1 such that $m_1 <_I m_2$. For example, if p sends message m_1 to q and then sends m_2 to r, it appends m_1 to m_2. If, after receiving m_2 appended with m_1 from p, r sends m_3 to s, then r appends m_1 and m_2 to m_3. An arbitrary protocol is easily transformed into a total-communication protocol by padding messages and extending states in the obvious fashion and ignoring all appended messages.

In Theorem 2 below we will establish certain necessary properties of WT-TC protocols in which no processor becomes amnesic. In doing so, we will consider only protocols with no $\overline{E}$ states. We now justify this behavior.

Let P be a total-communication protocol. A processor in P may "receive" a message in two ways: directly, from the source, or indirectly, by receiving a copy of the message appended to some other message. Let P' be the total-communication protocol obtained from P as follows. Each processor p' in P' simulates one processor p in P. The basic idea will be to eliminate $\overline{E}$ states by forcing the simulated processor to process each message as soon as possible once its existence is known. Since P is a total-communication protocol, learning of a message's existence is equivalent to receiving a copy of the message.

p' may receive copies of several messages within a given transmission, but p can only process one message at each of its steps. p' therefore has a special set of *simulated receiving* states, as well as a special buffer for holding unprocessed messages. The buffer is maintained as a priority queue. In simulating any transition to a receive state p' first enters a simulated receiving state and checks the priority queue. If the queue is empty, p' enters a true receiving state. If it is not empty, p' simulates receipt of the first message on the queue, and processes that message. All processed messages are saved and marked "old." Whenever a transmission is received, p' breaks it into separate messages and inserts these into the priority queue, discarding all duplicate copies and all newly received copies of messages already processed and marked "old."

Recall that processors are not permitted to send messages to themselves. Thus, the only way p could enter an $\overline{E}$ state is by receiving a message m from a processor that knows of an undelivered message m'. As we have observed, this means that p actually receives a copy of m' indirectly. Since m' was sent before m was sent, p will insert m' ahead of m in the priority queue. Thus, when p' simulates p's transition on message m, p will have already processed m' in the simulation, so p' will not simulate a transition to an $\overline{E}$ state.

Note that if processors can become amnesic then we can't use the same trick to avoid $\overline{E}$ states. For example, a processor p may become amnesic only if it knows that at least one of a set of possible messages, say m, has been sent to q. q may then enter an $\overline{E}$ state by querying p and learning that p is amnesic. On the other hand, p can't send q a copy of m when queried because it is amnesic. Finally, if the set of possible messages has cardinality at least two, then q cannot deduce the text of m from the fact that p is amnesic.

4. Six Consensus Problems

In this section we study the relationships among the six problems obtained by combining each of the consistency conditions (interactive and total) with the three termination conditions (weak, strong, and halting). We assume fail-stop processors and a decision rule of unanimity. We specify a problem by specifying its consistency and termination conditions. For example, WT-TC denotes the weakly terminating total consistency problem. The first theorem follows immediately from the definitions.

Theorem 1: For any termination condition T $\in$ {WT, ST, HT}, T-IC $\leq$ T-TC. For any consistency constraint C $\in$ {IC, TC}, WT-C $\leq$ ST-C $\leq$ HT-C.

Proof: Total consistency implies interactive consistency, since if no two processors disagree then certainly no two operational processors disagree. Thus, any protocol establishing T-TC also establishes T-IC, so by definition T-IC $\leq$ T-TC. Similarly, halting termination implies strong termination, which in turn implies weak termination, so WT-C $\leq$ ST-C $\leq$ HT-C. □

We now give certain necessary properties of protocols for WT-TC. We will use these properties and Theorem 1 to show formally that none of the consensus problems are equivalent.

A state s ***implies*** predicate X if X holds in every accessible configuration containing s.

Definition: A state s is ***safe*** if and only if it satisfies both:

(1) $C(s)$ contains at most one decision state; and

(2) if $C(s)$ contains a commit state, then s implies that the input value of each processor is "1".

Theorem 2: Let P be a WT-TC protocol with no $\overline{E}$ states. Then all states of P are safe.

Before proving this theorem, let us first consider intuitively why it is true. A nonfaulty processor in a TC protocol must be able to decide in accordance with all other decided processors, even when they have failed. Consider the case in which all processors but one, p, fail. p may have to base its decision solely on its state. Thus, in some cases p's state must imply that at least one type of decision was ***not*** made by another processor. Furthermore, if p's concurrency set contains a commit state then p may be forced to commit. Since p can commit only if the initial bits of all processors are 1, p's state must imply this condition. Note that these are the requirements of a safe state.

The proof of Theorem 2 requires the following technical definition and lemma.

Definition: Let X be any set of processors and C be any configuration. We let ***state***(X, C) denote the projection of C onto the states of all processors $p \in X$.

Lemma 3: Let C and D be configurations and X a set of processors such that ***state***$(X, C) =$ ***state***(X, D). If σ is any finite sequence of steps applicable to both C and D, then ***state***$(X, \sigma(C)) =$ ***state***$(X, \sigma(D))$.

Proof: Let p be an arbitrary processor in X. We show by induction on the length of σ that p's state in $\sigma(C)$ is the same as its state in $\sigma(D)$.

The basis, ***length***$(\sigma)=0$, is trivial. Suppose the lemma holds for any schedule σ where ***length***$(\sigma)<n$ $(n>0)$. Consider now a σ with length n $(n>0)$. Note that σ is of the form $\sigma'\sigma_n$ where σ' is a schedule of length $n-1$ and σ_n is an event, By definition, $\sigma(E)=\sigma_n(\sigma'(E))$ for any configuration E.

By the induction hypothesis, we have $state(p, \sigma'(C))=state(p, \sigma'(D))$. We need to show that p's state in $\sigma_n(\sigma'(C))$ is equal to its state in $\sigma_n(\sigma'(D))$. There are two cases. In the first case, σ_n has the form (q,μ) where $q \neq p$. In this case, applying σ_n does not change the state of p. Hence, $state(p, \sigma_n(\sigma'(C)))=state(p, \sigma_n(\sigma'(D)))$.

In the second case, σ_n has the form (p, μ). Since p acts deterministically, it makes the same transition when σ_n is applied to $\sigma'(C)$ as it makes when σ_n is applied to $\sigma'(D)$. Again we have $state(p, \sigma_n(\sigma'(C)))=state(p, \sigma_n(\sigma'(D)))$. □

In Lemmas 4 and 5 we prove that any state of a WT-TC protocol without $\bar{E}$ states must satisfy, respectively, conditions (1) and (2) in the definition of a safe state.

Lemma 4: Let P be as in the statement of Theorem 2 and let s_p be any operational state in P. Then $C(s_p)$ contains at most one decision state.

Proof: Suppose the lemma is false. Then there exist configurations C_C and C_A containing s_p and containing commit and abort states, respectively. Clearly, s_p is not a decision state, since otherwise one of C_C and C_A would contain conflicting decisions. Let I_A be a finite execution resulting in C_A and let F_A denote the failed processors when the system is in C_A. Let C_A' be the configuration reached from C_A by applying failure events for all processors in $F_C - F_A$, and let I_A' be the corresponding extension to I_A. Similarly, let I_C be a finite execution resulting in C_C and let F_C denote the failed processors when the system is in C_C. Let C_C' be the configuration reached from C_C by applying failure events for all processors in $F_A - F_C$, and let I_C' be the corresponding extension to I_C. (Note that the same failures occur in I_A' as occur in I_C'.) Clearly $state(p, C_A') = state(p, C_C') = s_p$. Now, s_p contains a record of all failures detected by p during both I_A' and I_C'. Thus, during I_A', p is notified of the failure of some processor q if and only if p is so notified during I_C'.

Let σ be any finite deciding run applicable to C_A' in which the only messages received by p are failure notices. Such a run exists because all the remaining operational processes but p may fail immediately and p must still be able to decide. Then σ is applicable to C_C and by Lemma 3 the same decision must be reached in both cases. Thus either $\sigma(C_A')$ or $\sigma(C_C')$ contains conflicting decisions. □

Lemma 5: Let P be as in the statement of Theorem 2 and let s_p be any state in P. If $commit \in C(s_p)$, then s_p implies satisfaction of the commit rule.

Proof: Let C be a configuration containing s_p and a commit state. Let σ be any finite deciding run applicable to C beginning with the failure of all processors except p and in which the only messages received by p are failure messages. Then p must commit in $\sigma(C)$, whence, by the correctness condition, the commit rule must be satisfied. □

Together, Lemmas 4 and 5 imply Theorem 2.

We say a state s is *committable* if and only if s implies that all initial bits are 1 and $C(s)$, the concurrency set of s, contains no abort state. Otherwise we say s is *noncommittable*. This partition of the state set determines the *bias* of a state.

The following corollary is immediate from Theorem 2.

Corollary 6: In any total consistency protocol establishing even weak termination, if a processor has decided then every nonfaulty processor shares its bias.

Let us call a protocol *safe* if all its operational states are safe. Note that a safe protocol need not be a WT-TC protocol, in fact, it can be the trivial protocol in which processors have input and decision registers but do nothing. Let us call a configuration *safe* if it is the result of a finite execution of a safe protocol. The next theorem shows that WT-TC can always be reached from a safe configuration.

Theorem 7: From any safe configuration in which at least one processor occupies a sending state it is always possible to establish WT-TC within $O(N^2)$ steps per processor, where N is the number of processors in the system.

The proof of this theorem requires the construction of a "termination protocol" that can take as its initial configuration an arbitrary safe configuration and then establish WT-TC within the indicated bounds. Since WT-TC termination protocols have appeared at least twice in the literature ([S81], [S82]), we omit the formal proof of this theorem. One such protocol appears in the appendix.

We are interested in Theorem 7 primarily because it allows us to work with partial specifications of WT-TC protocols. In the proofs that follow, we will only specify the failure-free behavior of WT-TC protocols. Whenever a failure occurs, the termination protocol will complete the execution.

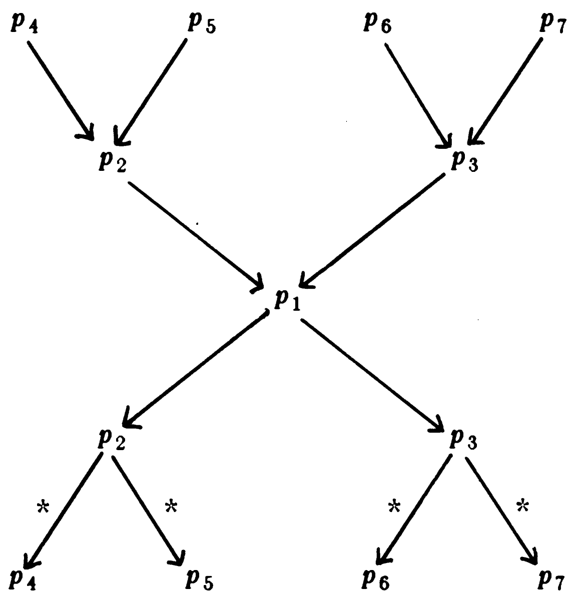
The next theorem shows that HT-IC and WT-TC are incomparable. There exists a protocol ensuring the strongest termination condition and weaker consistency condition which cannot guarantee the stronger consistency constraint, even under the weakest termination condition. Conversely, there is a protocol for WT-TC which cannot guarantee the weaker consistency constraint under halting termination.

Theorem 8: HT-IC and WT-TC are incomparable.

Proof: We first prove that HT-IC does not reduce to WT-TC. Consider the WT-TC protocol for 7 processors presented in Figure 1. Only the failure-free behavior is described; whenever a failure is detected processors invoke the termination protocol given in the appendix.

Although the protocol solves WT-TC, it cannot solve HT-IC. To see this, suppose that p_4 sends "0" as its input value. Then p_4 knows all processors are noncommittable and they will retain that bias, so p_4 can abort and no further messages will be sent to it. The communication pattern in which one processor halts after sending a single message and receiving none cannot be the communication pattern of any protocol for HT-IC. Suppose it were. If a processor receives no messages, then it cannot know input values of the other processors. Thus, if a processor halts without receiving any messages, then it halts in an abort state. We describe two scenarios, indistinguishable to p_6. In one scenario p_4 halts in an abort state, in the other it halts in a commit state.

Scenario 1: p_6 sends a "1" as its input value; p_4 send "0" as its input value and halts in an abort state without receiving further messages. All processors but p_4 and p_6 fail before p_3 sends to p_6 in Phase 1. Not only is p_6 undecided, but it doesn't know if p_4 is undecided or halted in an abort state. Thus, p_6 cannot wait for a message from p_4.



(a) The communication scheme for a phase. (*No message sent to a leaf with an input of 0.)

PHASE 1.

send inputs toward root (p_1);
root sets ***bias*** according to values of all inputs;
root sends ***bias*** toward leaves (no message sent to
leaf with input 0);
if ***bias*** = noncommittable processor aborts and
Phase 2 is omitted;

PHASE 2 (executed only if ***bias*** = committable).

after receiving ***bias***, each leaf sends an
acknowledgement toward root;
after receiving all acknowledgements, root sends
commit toward leaves;

(b) An informal description of a WT-TC tree protocol.

Figure 1. A WT-TC protocol that can not solve HT-IC. The protocol uses a tree communication scheme.

Scenario 2: All processors send "1" for their input values. p_4 becomes committable and begins phase 2. All processors but p_4 and p_6 fail. p_4 does not know if p_6 is noncommittable or has actually committed and halted, so p_4 must commit without waiting for a message from p_6.

Thus, there exist configurations C_C (scenario 1) and C_A (scenario 2) such that $\mathit{state}(p_6, C_C) = \mathit{state}(p_6, C_A)$ and C_C and C_A contain commit and abort states, respectively. Consider any finite deciding run σ applicable to C_A. Clearly, p_6 must abort in $\sigma(C_A)$. Since σ contains only failure messages (p_4 does not send any messages because it has halted), σ is also applicable to C_C. By Lemma 3, p_6 must abort in $\sigma(C_C)$ as well, violating IC.

It remains to show that WT-TC does not reduce to HT-IC. The protocol presented in Figure 2 solves HT-IC but does not solve WT-TC. This is because p_0 decides before all nonfaulty processors share its bias and halts without receiving any further messages. It therefore violates Corollary 6 whenever the decision is to commit. □

From Theorem 8 and its proof, it follows that for a given termination condition, the IC problem and the TC problem are not equivalent: the set of protocols solving IC is richer than the set solving TC.

Corollary 9: For all termination conditions T $\in$ {WT, ST, HT}, T-IC $<$ T-TC.

Proof: We need only show strictness, since reducibility is a result of Theorem 1. Assume for the sake of contradiction that T-TC $\leq$ T-IC. By Theorem 1, we have WT-TC $\leq$ T-TC for any T $\in$ {WT, ST, HT}. Similarly, we have T-IC $\leq$ HT-IC. Hence, WT-TC $\leq$ T-TC $\leq$ T-IC $\leq$ HT-IC, which implies WT-TC $\leq$ HT-IC and thereby contradicting Theorem 8. □

In addition, Theorem 8 implies

Corollary 10: For all consistency conditions C $\in$ {IC, TC}, WT-C $<$ HT-C.

The proof is similar to the proof of Corollary 9 and is omitted.

Corollary 11: HT-IC and ST-TC are incomparable.

Proof: That HT-IC does not reduce to ST-TC follows from the observation that if the protocol of Figure 1 is modified so that processors become amnesic as soon as they decide then we obtain a protocol for ST-TC. (The termination protocol is modified by having processors broadcast the fact that they are amnesic as soon as they detect a failure. Amnesic processors are then deleted from the *UP* sets of the other processors.)

To show that ST-TC does not reduce to HT-IC, suppose the opposite. Now, ST-TC $\leq$ HT-IC (by assumption) and WT-TC $\leq$ ST-TC (by Theorem 1); hence, WT-TC $\leq$ HT-IC (by transitivity of $\leq$). This, however, contradicts Theorem 8. □

The above implies the next corollary, whose proof is similar to that of Corollary 9.

Corollary 12: For all consistency conditions C $\in$ {IC, TC}, ST-C $<$ HT-C.

We can also prove that under either consistency constraint weak termination differs from strong termination.

Notes

(1) The communication primitive "broadcast (*message*, *set-of-processors*)" sends *message* to each processor in *set-of-processors* (order unspecified).

(2) The communication primitive "receive_all (*set-of-processors*)" delays the processor until a message from each processor in the set is received. It returns a set of messages, one from each process.

p_0:

  Msgs := receive_all(P−{p_0});

  if no failures detected →

    compute *decision* based on *Msgs* and $input_0$

  [] failures detected → *decision* := abort

  fi;

  broadcast(*decision*, P−{p_0});

  decide;

  halt

p_i ($1 \leq i \leq N-1$):

  send(input v_i, p_0);

  decision := receive();

  if no failures detected →

    broadcast(P−{p_0, p_i},*decision*);

    decide

  [] failures detected →

    call modified termination protocol

  fi;

  halt

The termination protocol is modified as follows: Whenever a processor receives a decision message, it removes the sender from its *UP* set (the sender has halted). Except for this, decision messages are classified as committable/noncommittable and processed as usual.

Figure 2. An HT-IC protocol not solving WT-TC.

Theorem 13: For every consistency constraint C $\in$ {IC, TC}, WT-C $<$ ST-C.

Proof: We need only show strictness. To see that WT-IC $<$ ST-IC, consider the following WT-IC protocol. Each p_i, $1 \leq i < N$ begins by sending its input to p_0. p_0 tallies the inputs, including its own, decides, and sends a decision to p_1. p_1 decides accordingly and forwards the decision to p_2, and so on, until the decision reaches p_{N-1}, which simply decides. The communication pattern for this protocol is illustrated in Figure 3. The pattern illustrated is the only failure-free pattern of the protocol. This communication pattern cannot handle both decisions to commit and to abort in an ST-IC protocol. Suppose otherwise. Then each processor p_i sending "1" as its input must become amnesic after deciding, without receiving further messages. Consider the following two scenarios.

Scenario 1: p_0 and p_2 send "1", p_0 commits and becomes amnesic, and p_1 and p_3 fail before the decision message is sent to p_2.

Scenario 2: p_0 and p_2 send "1", but p_1 sends "0". p_0 aborts and becomes amnesic, and p_1 and p_3 fail before the decision message is sent to p_2.

By an argument similar to the proof of Lemma 3, p_2 must reach the same decision in each case, so in some execution p_0 and p_2 reach mutually inconsistent decisions.

We now show that WT-TC $<$ ST-TC. This result is considerably less intuitive and the proof is very contrived.

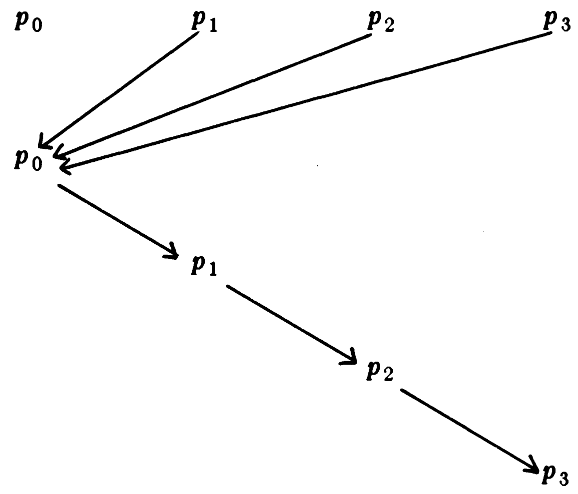


Figure 3. A WT-IC protocol that can not solve ST-IC.

Consider the WT-TC protocol P with four failure-free communication patterns, as represented in Figure 4.

The figure shows 2 kinds of edges. Solid edges are messages that are sent in every failure-free execution of the protocol. Dashed edges represent messages that are sent or not sent according to the order in which certain other messages are delivered. In particular, message m_1 is sent only if m_a is delivered before m_b is delivered. Message m_2 is sent only if m_c is delivered before m_d. Finally, m_3 is sent only if both m_1 and m_2 are sent. Thus Figure 4 represents four possible communication patterns, according to which of the messages corresponding to dashed edges are sent: (1) none of m_1, m_2, m_3 are sent; (2) only m_1 is sent; (3) only m_2 is sent; and (4) m_1, m_2, and m_3 are sent.

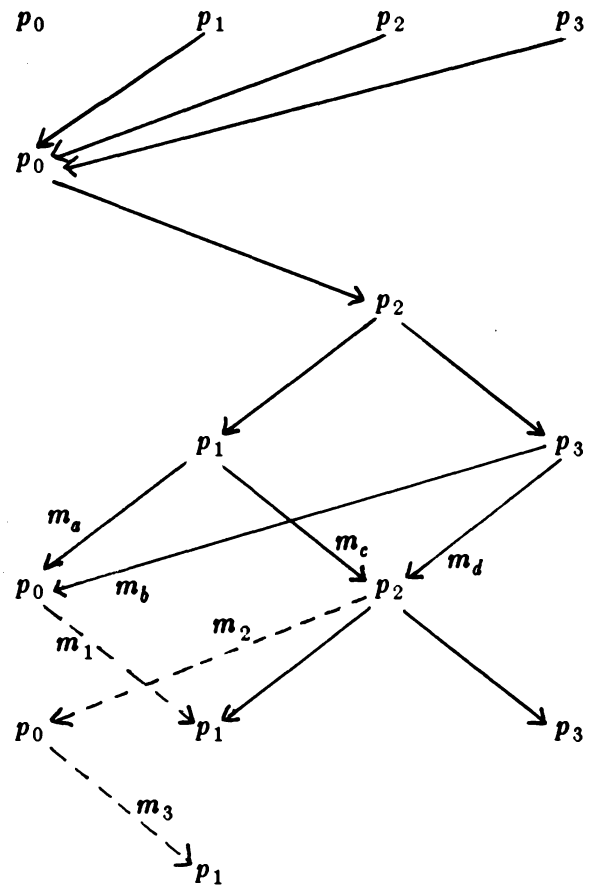


Figure 4. A WT-TC protocol that can not solve ST-TC.

The perversity of this example is that the messages corresponding to the dashed edges serve no purpose; indeed, eliminating these edges leaves us with the a perfectly good communication pattern for both a WT-TC and an ST-TC protocol.

Let us assume for the sake of contradiction that the scheme of P is the scheme of an ST-TC protocol. Then there exist an execution in which m_1 is sent and an execution in which m_1 is not sent in both of which p_0 becomes amnesic before receiving m_2. This is obvious, since m_2 might never be sent and p_0 must become amnesic eventually. Consider two executions, I_y and I_n, such that p_0 sends m_1 in I_y and p_0 does not send m_1 in I_n and such that in both executions p_0 becomes amnesic and then receives m_2 and p_0's state on receipt of m_2 is the same in both executions. Since p_0 behaves deterministically it must send m_3 in both executions or neither. If neither, then the communication pattern of I_y is not one of the patterns (1)-(4) listed above. If both, then the communication pattern of I_n is not one of the patterns (1)-(4) listed above. This contradicts our assumption that the scheme of P is the scheme of an ST-TC protocol. □

The following diagram summarizes the results of Theorems 1, 8, and 13, and Corollaries 9 through 12. Notice that all of the inequalities are strict.

$$\begin{array}{ccc} \text{WT-IC} & < & \text{WT-TC} \\ < & & < \\ \text{ST-IC} & < & \text{ST-TC} \\ < & & < \\ \text{HT-IC} & < & \text{HT-TC} \end{array}$$

References

[BT] Bracha,G. and Toueg, S. "Resilient Consensus Protocols." *Proc. 2nd ACM Symposium on Principles of Distributed Computing* (1983), 12-26.

[DDS] Dolev,D., Dwork, C., and Stockmeyer, L. "On the Minimal Synchronism needed for distributed Consensus." *Proc. 24th IEEE Symposium on Foundations of Computer Science* (1983), 393-402.

[DFFLS] Dolev, D., Fischer, M., Fowler, R., Lynch, N., Strong, H. R., "An Efficient Byzantine Agreement Without Authentication." *Information and Control*

[DLPSW] Dolev, D., Lynch, N., Pinter, S., Stark, E. and Weihl, W. "Reaching Approximate Agreement in the Presence of Faults." *Proc. 3rd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems* (1983).

[DLS] Dwork, C., Lynch, N., and Stockmeyer, L. "Consensus in the Presence of Partial Synchrony." *These proceedings.*

[DRS] Dolev, D., Reischuk, R., and Strong, H.R. " 'Eventual' is Earlier Than 'Immediate'." *Proc. 23rd IEEE Symposium on Foundations of Computer Science* (1982), 196-203.

[DS] Dolev, D. and Strong, H. R. "Distributed Commit with Bounded Waiting." *Proc. 2nd Internation Symposium on Distributed Data Bases* (1983, 53-60.

[F] Fischer, M. "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)." YALEU/DCS-/RR-273, June, 1983.

[FLP] Fischer, M., Lynch, N. A. and Paterson, M. "Impossibility of Distributed Consensus with One Faulty Process." *Proc. 2nd Symposium on Principles of Database Systems* (1983), 1-7.

[GPD] Garcia-Molina, H., Pittelli, F., and Davidson, S. "Is Byzantine Agreement Useful In A Distributed Database?" *Proc. 3nd Symposium on Principles of Database Systems* (1984), 61-69.

[Gi] Gifford, D.K. "Weighted Voting for Replicated Data." Technical Report CSL-79-14, (1979), XEROX Palo Alto Research Center.

[Gr] Gray, J. "A Discussion of Distributed Systems." Research Report RJ2699 (1979), IBM.

[L78] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *CACM 21,* 7 (1978), 558-565.

[L83] Lamport, L. "The Weak Byzantine Generals Problem." *J. ACM 30,* 3 (1983), 668-676.

[LPS] Lamport, L., Shostak, R., and Pease, M. "The Byzantine Generals Problem." *ACM Trans. on Programming Lang. and Systems 4,* 3 (1982), 382-401.

[ML] Mohan, C., and Lindsay, B., "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," *Proc. 2nd ACM Symposium on Principles of Distributed Computing* (1983), 76-88.

[PSL] Pease, M., R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults." *J. ACM 27,* 2 (1980), 228-234.

[S] Schneider, F. B. "Byzantine Generals in Action: Implementing Fail-Stop Processors." *ACM Trans. on Computer Systems 2,* 2 (1984), 145-154.

[SGS] Schneider, F.B., Gries, D., and Schlichting, R. D. "Fast Reliable Broadcasts." Computer Science Technical Report 82-519 (1982), Cornell University.

[S81] Skeen, D. "A Decentralized Termination Protocol." *Proc. 1st IEEE Symposium on Reliability in Distributed Software and Database Systems* (1981), 27-32.

[S82] Skeen, D. "Crash Recovery in a Distributed Database System." Technical Report UCB/BRL M82/45 (1982), Department of Electrical Engineering and Computer Science, University of California, Berkeley.

[W] Wensley, J.H., et al. "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control." *Proc. IEEE 66,* 10 (1978), 1240-1255.

Appendix: A Termination Protocol

The protocol below ensures total consistency and establishes weak termination when invoked from any configuration in the execution of a safe protocol. The code given is for an arbitrary processor p.

```
protocol Termination (bias_p, UP_p);
local variables Msgs: set of messages;
                round: 1..N;

for round := 1 to N do
    broadcast (UP_p-{p}, (round, bias_p));
    Msgs := receive_all(UP_p-{p}) modified so that
         messages from this round only are received;
    UP_p := UP_p - {q | "failed(q)" received};
    if "committable" received
       then bias_p := committable;
    fi;
od;
if bias_p =committable → commit
 [] bias_p =noncommittable → abort
fi;
halt
```

Notes.

(1) The communication primitives "broadcast" and "receive_all" are defined in Figure 2.

(2) The global variable N contains the number of participating processors.

(3) The parameters are two components of the state of p in the consensus protocol invoking this termination protocol: $bias_p$—indicating committable or noncommittable, and UP_p—the set of processors whose failures have not be detected by p.