Collective Consistency

Cynthia Dwork, Ching-Tien Ho, Ray Strong

IBM Almaden Research Center

Abstract. We present *collective consistency*, a new paradigm for multi-process coordination arising from our experience building a parallel version of a software package for computer aided engineering. A straightforward checkpointing and reorganization strategy will permit a parallel computation to tolerate failed or slow processes if the failure or tardiness is consistently detected by all participants. To this end, we propose the addition of *autonomous failure detection* and a *consistent reporting* protocol to the collective communication library supporting the industry standard Message Passing Interface. The focus of this paper is the consistent reporting protocol that must provide an approximate solution to the collective consistency problem. We give a family of knowledge-based specifications for a collective consistency protocol and discuss several implementations. We support our implementation choices with an extremely general proof that no trivially dominating solution exists.

1 Introduction

In a large body of parallel mathematical and engineering software computations have a regular structure of alternating communication and computation phases. The communication phase is carried out using the industry standard Message Passing Interface (MPI). In a typical implementation of the MPI, each participant exchanges multiple data packets with the others via a transport layer. A *blocking* (waiting) receive call is issued to the transport layer for all the packets that are to be received from other participants. If one participant fails to send all of its packets to the others, each member blocks (waits forever) pending the arrival of these anticipated packets. A blocked process itself fails to send packets during the subsequent communication phase, causing still other processes to block. Thus, in real computations, when even one process fails *the entire computation becomes blocked pending human intervention.* We do not know of any automatic method currently in use to address this problem.

We develop a method by which such failures can be automatically tolerated by the remaining processes and by which these processes can agree on the identities of failed processes. Our technique allows the processes to agree on a new group membership so that the computation can be reorganized to run on the new group. Of course, such agreement is impossible in a completely asynchronous system: any agreement protocol that guarantees consistency has executions in which some processes must block or take infinitely many steps [DDS, FLP]. However, one goal that *is* both desirable and possible to achieve is the reduction of

the window (time period) during which a process failure can cause some or all other processes to block. Specifically, we propose the following approach:

1. Augment the collective calls to the transport layer with a simple failure detector [CT, CHT], so that no process blocks during the collective communication.
2. Run a (preferably simple and brief) *collective consistency* protocol (CCP), during which some participants may block, allowing those processes that return to share a consistent view of the set of non-failed processes. Specifically, a collective consistency protocol ensures that any two processes that do not mutually regard each other as having failed and that do not block will have an identical view of group membership. (A formal definition of the problem appears in Section 2.)
3. Reorganize the application computation to run on the newly agreed-upon group. Continue the computation protocol, ignoring messages from the detected faulty processes.

Our target application is particularly amenable to this approach because each successful completion of a collective communication call presents a natural checkpoint, after which each process can easily recover sufficient information to reorganize and proceed with the computation. Thus when a failure is consistently reported to all surviving participants, the survivors can roll back to the previous check point, reorganize, and roll forward. However, as long as the participants have different views of the set of processes available to participate after reorganization, no further progress is possible. Thus collective consistency is essential to fault tolerance in this setting.

As described informally above and defined formally in Section 2, *collective consistency* is a weak form of consensus in which processes try to reach a common view of group membership under a rather relaxed definition of "commmon." More specifically, a process enters the computation with an *initial view* consisting of a list of suspected failed processes, and, if it returns from the CCP, it does so with an *output view* of suspected faulty processes. The collective consistency requirement is that each process p's output view V_p must agree with the output views of processes that p does not suspect to have failed. In other words, p's ouput view need only agree with the output views of processes not in V_p. Since a symmetric requirement exists for each $q \neq p$, this means that the output views of p and q can differ only if they are mutually suspecting processes.

Thus, a collective consistency protocol partitions the survivors into cliques $C_1, \ldots, C_k$, where the processors in each clique C_i all believe that all processors not in C_i have failed, and each $p_j \in C_i$ has no reason to believe any other process in C_i has (yet) failed. The key difference between collective consistency and consensus is that the definition of collective consistency permits two returning processes, p and q, to have different views under some circumstances. By contrast, in consensus p's output can differ from q's only if at least one of them fails to return. A second difference is that the definition of collective consistency does not require all nonfaulty processes to terminate.

In Section 4 we give a family of knowledge-based specifications for protocols for collective consistency. We prove that certain implementations of these specifications solve the collective consistency problem and we provide two families of such implementations.

Collective consistency is weaker than agreement. Our situation is unlike the typical situation for asynchronous impossiblity proofs: it is not the case that any protocol for collective consistency has executions in which every process takes infinitely many steps. However, we prove the existence of executions E such that *either* every process takes infinitely many steps *or* there exists a particular process p_E such that in E *all* other processes block waiting for p_E. Thus, even in the presence of a single slow or crashed process, there is no non-blocking CCP. Indeed, the lower bound also rules out all hope of designing a single failure tolerant, nontrivial CCP in which, for example, at least one process terminates. This is discussed further in Section 5.

Our target application is computation intensive and certain parts, for example, those involving matrix multiplication, lend themselves easily to parallelization. The work described in this paper arose from our experience in building a prototype parallel implementation of this application, but the results apply to any parallel architecture in which multiple processing elements operate concurrently to perform a single task.

2 Requirements for Collective Consistency of Failure Detection

In this section we define several variants of collective consistency and discuss some design requirements for collective consistency protocols specific to our target applications.

2.1 Definitions of Collective Consistency

The goal of collective consistency is for the participants to reach a consistent view of the set of failed processes. In a collective consistency protocol there is no requirement that a process ever return. Thus, collective consistency can be solved in systems that do not permit solutions to consensus. The question of how to evaluate a collective consistency protocol is a subject for future research. We return to this question in Section 6.

Each process enters a CCP with an *initial view*, which is a list of processes it suspects of failure. (We treat the receipt of the initial view as the first event of the CCP for each process.) In the definition of collective consistency we need not concern ourselves with how these suspicions are generated, but in practice, suspicion is based on the output of a failure detector. During the course of the computation a process may irreversibly decide on an *output view* V – a final list of suspects. In this case we say the process *returns* V. A process that does not decide on an output view is said not to return.

The key requirement of collective consistency essentially says that if p does not regard q as having failed, and if indeed q has not failed and actually returns an output view, then this view is the same as p's:

Collective Consistency: If V_i is the view returned by process i and V_j is the view returned by process j, and if $j \notin V_i$, then $V_i = V_j$.

In other words, since the requirement is symmetric for i and j, if process i does not view process j as having failed, then they must agree on which processes they suspect of failure. We will use the term *weak collective consistency* as a synonym, emphasizing the difference between this property and that of *strong collective consistency*, defined next. Note that strong collective consistency is just consensus without termination.

Strong Collective Consistency: The returned views of all processes are identical.

A *quorum system* is a collection of sets of processes, every pair of which have a non-empty intersection [Gif, PW, NW]. Quorum systems can be used to convert a weak collective consistency protocol into one for strong collective consistency. As explained in the Introduction, a weak collective consistency protocol partitions the survivors into cliques such that, roughly speaking, the members of each clique believe that they, and only they, are alive. To obtain a protocol for strong collective consistency we add the following test: if a process sees that its clique does not contain a quorum then it blocks. To see that the transformation is correct, suppose two processes p and q complete the protocol. Let Q_p be a quorum contained in p's clique, and let Q_q be a quorum contained in q's clique. By definition of a quorum system, there exits a process $r \in Q_p \cap Q_q$. It follows from the definition of weak collective consistency that $V_p = V_r = V_q$. In our target application, each clique continues the computation independently. Under strong collective consistency, at most one clique of processes continues the computation.

In our target application, process q is in the initial view of process p (that is, before the CCP begins p suspects q of having failed), precisely because p did not receive from q an anticipated message containing information essential for p to continue the application computation. While under some circumstances it may be possible for p to obtain this information from other processes, this would be neither simple nor quick. From an engineering point of view, it is easier for all processes in p's group to regard q as having failed and to reorganize the computation accordingly. For this reason we are particularly interested in *monotone* solutions to the collective consistency problem, in which suspicions are never allayed – if p initially suspects q, then nothing during the CCP "convinces" p to stop suspecting q. This motivates the following definition.

Monotonicity: If V_i is the initial view of processor i, and if processor i returns V, then $V_i \subseteq V$.

Finally, we rule out the trivial solution to weak collective consistency in which every terminating process simply views every other as having failed, since this results in a trivial partition (every surviving process forms its own group consisting only of itself) and hence unecessary duplication of work in the target application. To make this precise we need a few definitions.

We model a protocol execution as sequences of events, one sequence for each process. A *cut* is the union of a set of finite prefixes of sequences, one for each process, closed under causality. The *initial cut* is the empty cut. Intuitively, the initial cut "happens before" processes obtain their initial views. This is merely a notational convenience.

Nontriviality: The initial cut c_0 is *multivalent*, that is, there exist cuts d_1 and d_2 containing c_0, such that in d_1 some process p returns an empty output view and in d_2 some (not necessarily different) process q returns a non-empty output view.

Let $\overline{\{i\}}$ denote the set of all processes except i. The nontriviality condition rules out the trivial "solution" to collective consistency in which each process i returns output view $V_i = \overline{\{i\}}$.

A protocol solves (strong) (monotone) collective consistency if it is nontrivial and in all executions the (strong) (monotone) collective consistency condition is satisfied.

2.2 Design Goals

In Section 3 we discuss the relationship between collective consistency and the widely studied group membership problem. The source of the problem, that is, the application in which the problem arises, naturally drives one's view of what constitutes a "good" solution. Much (but by no means all) of the work on group membership arises in the context of replicated databases. As mentioned above, collectively consistency arose in the context of parallel engineering and mathematical software. In this environment there is no danger if the processes split into two groups and each group pursues the computation independently (although of course this results in wasted effort). The same is typically not true in the context of a replicated database.

Our first desideratum is *simplicity*, based on the assumption that a simpler algorithm is more likely to be implemented correctly.

The second desideratum is *speed* (roughly speaking, minimizing the number of protocol steps a process must take before it sends the last message required of it by the protocol), since the system is vulnerable to blocking during execution of the CCP. Intuitively, there appears to be an engineering tradeoff in designing a collective consistency protocol: on the one hand, since failure detection is not performed during the execution of the collective consistency protocol, the more (asynchronous) rounds of communication the greater the chance that a process will block trying to complete a communication round. This would seem to suggest that a collective consistency protocol with few rounds of communication should "perform better" than a protocol requiring many rounds. On the other hand, a more complicated protocol can sometimes permit more processes to return. For example, if a process p requires information from process q, and if a third process r has received this information from q, then r can relay this information to p, even if q does not succeed in doing so. In this case p has had to wait two rounds for the information (one round for it to go from q to r and one round for

it to travel from r to p), but p has the information it needs and can return from the protocol.

The advantage to having more processes return is that more processes are then available to complete the application computation. A formalization of this intuition and a thorough analysis of the factors involved in choosing the "right" implementation of collective consistency would be interesting theoretically and extremeley useful in practice. This is an excellent area for future research, especially since this type of tradeoff is not restricted to collective consistency.

3 Related Research

The overall goal of our work is to run a parallel application in the presence of process failure or unacceptable slowness. The study of concurrent execution of work in the presence of failures in a message-passing distributed system was initiated by Bridgeland and Watro [BW]. They consider a system of t asynchronous processes that together must perform n independent units of work. The processes may fail by crashing and each process can perform at most one unit of work during the course of the computation. Bridgeland and Watro provide tight bounds on the number of crash failures that can be tolerated by any solution to the problem. Thus, the goal in the [BW] work is to design a protocol that guarantees that the work will be performed in every execution of the protocol.

Dwork, Halpern, and Waarts designed algorithms for t processors to perform n units of work provided even one of the t processors remains operational [DHW]. They focused on effort-optimal algorithms, in which the sum of the number of messages sent and the number of tasks actually performed (including tasks performed multiple times because a processor fails before being able to report its progress) was $O(n+t)$, which is asymptotically optimal, but the (synchronous) time costs are exhorbitant [DHW]. Faster algorithms were obtained by De Prisco, Mayer, and Yung [DMY]. The approach of Dwork, Halpern, and Waarts yields as a corollary a message-efficient agreement algorithm, improved upon by Galil, Mayer, and Yung [GMY].

Study of another related problem was initiated in a seminal paper by Kanellakis and Shvartsman [KS]. Specifically, they introduce the *Write-All* problem, in which n processes in a shared-memory system cooperate to set all n entries of an n-element array to the value 1. Kanellakis and Shvartsman provide an efficient solution that tolerates up to $n-1$ faults, and show how to use it to derive robust versions of parallel algorithms for a large class of interesting problems. Their original paper was followed by a number of papers that consider the problem in other shared-memory models (see [AW, BR, KS2, KPRS, KPS, MSP]).

There is an extensive literature on the related area of *membership services* (see, for example, [CHT2, DMS, RB] for definitions and results in asynchronous systems). *Group membership* itself is a loosely defined term that generally describes the requirements on (1) the *series* of *views* held by a given process, of the membership of a group (within some universe of interest) as this membership changes over time; and (2) the manner in which the series of views obtained by different processes according to (1) are related. In contrast, collective consis-

tency is a "one-shot deal": (monotone) collective consistency is defined in terms of (inputs and) outputs in a *single* execution of a collective consistency protocol. In particular, in our target application, collective consistency is concerned with the problem of combining initial views of failed processes (provided by a failure detector) to form a collectively consistent output view of failed processes – and hence of group membership (the non-failed processes).

Finally, we note that any algorithm for consensus yields an algorithm for strong monotone collective consistency (and hence all other variants of collective consistency). Very briefly, the reduction is as follows. Run in parallel n independent executions of a single-source agreement protocol to agree on the input views of each process (the ith execution is to agree on the input view of the ith process). Note that it is a property of the single-source agreement problem that if the source remains non-faulty throughout the execution then the value decided upon is the initial value of the source. This is important for monotonicity. Recall that in the collective consistency problem each input view is a set of processes. Take the union of the agreed upon views to be the (common) output view.

Since collective consistency does not require termination, consensus is strictly more powerful than all variants of collective consistency defined in this paper.

In the patent literature, Dwork, Halpern, and Strong describe a solution to the problem of performing work in the presence of failures that is both fast and effort-optimal in the absence of failures, and is correct in all instances [DHS]. A different approach was taken by Whiteside, Freedman, Tasar, and Rothschild: the work is statically divided into a set of tasks, each of which is assigned redundantly to several processing elements. Results produced are subject to majority vote resolution. Beyond voting and the use of timeout to detect failures, there is no attempt to reach agreement among the processes [W].

Our work on collective consistency can be viewed as an adaptation of the general method of Dwork, Halpern, and Strong [DHS] to the environment of parallel computation, in which particular care is taken both to allow for inconsistent failure detection (different processes detect different failures) and to guarantee consistency of response (roughly, differences in detected failures are resolved) at the cost of a small chance of blocking. Intuitively, the chance of blocking is small because the consistency protocol requires few messages; a process would have to fail *during* the consistency protocol to cause blocking.

The collective consistency problem is an example of a coordination problem, like the distributed commit and the consensus problems. Solutions to both of these problems can be and have been used in an *ad hoc* way, under certain assumptions, to solve collective consistency. No solution, including those presented in this paper, is entirely general: each approach works only under certain assumptions about the environment. However, the solutions provided here seem to require the weakest assumptions. We elaborate by characterizing typical assumptions and, in the table below, associating each approach with its required assumptions.

The types of assumptions are

1. type of failure, *e.g.,* processes are restricted to crash failures;
2. number of failures, *e.g.,* $n \geq 3t + 1$;
3. timing, *e.g.,* there is an upper bound on message delivery time;
4. randomness, *e.g.,* each process has access to a private source of random bits; and
5. stable storage.

distributed commit	consensus	collective consistency
1 and 5	(1 or 2) and (3 or 4)	1

Each of the approaches discussed above either in the worst case makes no global decisions (no coordination) and allows all the work to be done by each of the participants [BW, W], or depends on the timing or randomization assumptions we associate with consensus above [DHW, KS]. Strong collective consistency has been studied before, e.g. by [CT, DLS, R]. Rabin called the problem *choice coordination*, and studied it in the context of shared memory with atomic test and set (the atomicity of test and set can be viewed as an additional timing assumption). The relevant results of Chandra and Toueg and Dwork, Lynch, and Stockmeyer discuss conditions under which consensus can be reached if the failure detectors [CT] or the system [DLS] remain sufficiently well behaved for sufficiently long. By contrast, our approach is to take input from failure detectors but make no further use of them during the collective consistency protocol. We know of no literature on monotone collective consistency, strong or weak.

4 Protocols for Monotone Collective Consistency

In this section we present a knowledge-based specification for a family of monotone collective consistency protocols. We give two implementations of this specification. The implementation assumes only crash process failures and that messages are not altered. We also assume (although it is not necessary for correctness) that each message sent eventually reaches its destination unless either the sender or receiver crashes.

In Figure 1 we give a knowledge-based specification for a protocol for monotone collective consistency of failure detection. The operator K_i in the specification is the knowledge operator, indicating that the truth of its argument is known to process i [FHMV].

We use the notation V_i^k to denote the value of V_i formed during iteration $k = 1, 2, \ldots$ by process i (V_i^0 is the initial value). If process i waits forever in iteration ℓ, then $\forall k \geq \ell$ we say V_i^k is *undefined*. The parameter B is an upper bound on the number of times the main loop in the code is iterated. Each value of B yields a different protocol.

In Figure 1, the knowledge operator K_i and its argument ($\phi = \forall j \notin V_i^{k+1}$, either V_j^{k+1} is undefined or $V_j^{k+1} = V_i^{k+1}$) represent an unkown Boolean predicate. An *implementation* of such a knowledge-based specification is an algorithm

Monotone-CC with inputs V_i and B:

$k = 0$;
$V_i^0 = V_i$;
Repeat for $k < B$\{
    Send V_i^k to all processors;
    $\forall j \notin V_i^k$, wait for V_j^k;
    Set V_i^{k+1} to be V_i^k union all V_j^k for $j \notin V_i^k$;
    If $K_i(\forall j \notin V_i^{k+1}$, either V_j^{k+1} is undefined or $V_j^{k+1} = V_i^{k+1})$
        then return(V_i^{k+1}) and exit;
    else $k = k + 1$;\}
Halt without returning;

Fig. 1. Monotone Collective Consistency Protocol; Code for Process i

that replaces $K_i(\phi)$ with a specific predicate. A *sound* implementation replaces $K_i(\phi)$ with a predicate Ψ such that, when Ψ returns True, ϕ holds in the set of possible executions provided by the system in which each participant uses Ψ in place of $K_i(\phi)$. An implementation is said to be *nontrivial* if (1) there is an execution in which each input is empty and after some number of iterations the Ψ at some process returns True and (2) there is an execution in which the input to some process p is nonempty and after some number of iterations the Ψ for p returns True.

Theorem 1. *Any nontrivial sound implementation of Protocol Monotone-CC provides a nontrivial solution to the monotone collective consistency problem.*

Proof. The proof is independent of the bound B on the number of iterations of the main loop, provided $B \geq 1$. In the remainder of the proof we assume $1 \leq k, \ell \leq B$.

Monotonicity is obtained because at all times V_i^k is the union of the initial V_i and other values. Nontriviality follows from monotonicity and the definition of a nontrivial implementation. We now argue collective consistency. Suppose processes p and q return V_p^k and V_q^ℓ respectively, and either $p \notin V_q^\ell$ or $q \notin V_p^k$. Assume without loss of generality that $q \notin V_p^k$. We first show that for all $r \notin V_p^k$, either r does not return or V_r^k is defined and $V_r^k = V_p^k$.

Let $r \notin V_p^k$. Since V_p^c only grows with c, we have that $r \notin V_p^c$ for all $0 \leq c \leq k$; that is, r was never suspected by p. Since p terminates with V_p^k we have that p successfully received V_r^c for all $0 \leq c < k$. If r does not complete the kth iteration of the loop, then it never returns. If r does return then it must complete the kth iteration, and in particular V_r^k is defined. By soundness, if V_r^k is defined, then $V_r^k = V_p^k$.

It follows from the previous argument and the fact that q returns with V_q^ℓ, that $\ell \geq k$ and hence that V_q^k is defined and equal to V_p^k.

Even if process q performs additional iterations, that is, even if $\ell > k$, it will only collect views from processes $r \notin V_q^k = V_p^k$, and all of these satisfy $V_r^k = V_p^k$. A simple induction shows that for all $k' \geq k$, if $V_r^{k'}$ is defined, then $V_r^k \subseteq V_r^{k'} \subseteq \cup_{r' \notin V_r^k} V_{r'}^k = V_p^k = V_r^k$, so $V_r^{k'} = V_r^k$. Thus, process q returns $V_q^\ell = V_p^k$.

One simple nontrivial sound implementation is obtained by letting $\Psi_1 = \forall j \notin V_i^k : (V_j^k = V_i^k)$. That is, Ψ_1 holds in the first iteration (*i.e.*, with $k = 0$) if, for all j not initially suspected of failure by i, j's initial list of suspects agrees with i's initial list of suspects. Letting $B = 1$ (*i.e.*, executing the loop with $k = 0$, returning if possible at the end of this first iteration, and otherwise not returning) Ψ_1 gives an extremely simple and yet nontrivial protocol for monotone collective consistency. We have designed a failure detector and communication transport layer so that if crash failures, significant slowdowns, or occasional message losses occur sufficiently far away in time from the execution of the collective consistency protocol, then failure detection will be uniform among those detecting a given failure. Thus the important window of vulnerability to failure for our simple Ψ is limited to a brief time interval surrounding execution of the collective consistency protocol. In fact, without iteration, the vulnerability window is limited to a brief time before the last failure detector of a machine that has not failed or slowed gives its input to its instance of the consistency protocol.

The *advantage* in iterating (that is, letting $B > 1$), is that it sometimes allows additional processes to terminate in spite of the occurence of failures in the failure window. We elaborate with an example. Since our failure detector uses a timeout, there are circumstances in which failure detection is not uniform. In particular there are executions in which some processes detect a slow process while others do not. Suppose we have a system of only three processes. Consider an execution in which there are no crash failures, but, based on its failure detector, process p suspects that process r has failed while process q does not. Using Ψ_1 with $B = 1$ *no* process returns from Monotone-CC. On the other hand, if we simply change to $B = 2$ then each of p, q, and r returns $\{r\}$ after the second iteration. In the context of our application, this is the desired outcome because p did not receive essential information from r in a timely fashion. The *risk* in iterating is that it increases the chances for a process failure (during execution of the collective consistency protocol) to cause other processes to block.

The predicate specified in Figure 2 provides another family of nontrivial sound implementations of the knowledge based protocol specified in Figure 1. The set W contains those processes j such that someone "trusted" by i (not viewed by i as faulty) "vouches" for j (does not view j as faulty). So in the above scenario this would cause p to wait to hear from r. Using Ψ_2 therefore allows this scenario to be handled even with $B = 1$. Although iteration seems to compensate for a difference in initial views, we conjecture that iteration is not necessary: the same effect can be obtained by a more sophisticated implementation (*e.g.*, using Ψ_2 instead of Ψ_1). Moreover, iteration is always dangerous, as it introduces additional opportunities for blocking.

Predicate Ψ_2:

$W = \{j | \exists m \notin V_i^{k+1} : (j \notin V_m^k)\}$;
$\forall j \in W$, wait for V_j^k;
$\forall m \notin V_i^{k+1}$, set $V_m^{k+1} = V_m^k$ union all V_j^k for $j \notin V_m^k$;
return$(\forall m \notin V_i^{k+1} : (V_m^{k+1} = V_i^{k+1}))$;

Fig. 2. Code for Predicate Ψ_2

As mentioned in Section 2.1, we can transform a protocol for weak collective consistency into one for strong collective consistency by adding a requirement that the complement of the output set be a member of a previously chosen quorum system. We can also use quorum systems to overcome a weakness of protocols that satisfy the knowledge-based specification of Figure 1: all such protocols block if some process fails at the beginning of the protocol because each other process waits to receive the view of the failed process. Let $\mathcal{Q}$ be a quorum system. In figure 1 replace the line "$\forall j \notin V_i^k$, wait for V_j^k;" with "Wait until $\{j | V_j^k$ received $\} \in \mathcal{Q}$;" and "Set $Q = \{j | V_j^k$ received $\}$;." Replace "$j \notin V_i^k$" in the next line by "$j \in Q$;." Then $\Psi_3 = \forall j \in Q : (V_j^k = V_i^k)$ can be used to provide a sound nontrivial implementation of the resulting specification. In fact it provides a solution for the monotone strong collective consistency problem.

5 A Lower Bound for Collective Consistency Protocols

As discussed above, we could not hope for a collective consistency protocol that tolerates process failure and guarantees termination for all correctly functioning participants; however, we had hoped it would be possible to design a collective consistency protocol that tolerated process failure and guaranteed termination for a nontrivial subset (possibly a minority) of all the correctly functioning participants. The following lower bound result shows that this is not the case: no collective consistency protocol can guarantee the timely (nonblocking) termination of any of the participants in the presence of one slow or failed process. Indeed, we show that, roughly speaking, for every collective consistency protocol either there exists a process p such that in some execution every process must wait to hear from p, or there is an execution in which no process fails but no process ever gives output.

5.1 Definitions and Axioms

We model a protocol execution as sequences of events, one sequence for each process. A *cut* is the union of a set of finite prefixes of sequences, one for each process, closed under causality. The first event at each process is an *input* event, the input being a possibly empty, finite set. Some processes may have later *output* events, at most one per process, the output being a possibly empty, finite set.

Recall that a cut c is *multivalent* if there exist cuts d_1 and d_2, both containing c, such that d_1 has a process that has given as protocol output the empty set and d_2 has a process that has given as protocol output some non-empty set. Note that the two outputs are inconsistent with respect to weak collective consistency. A protocol is *nontrivial* if the empty cut is multivalent. A cut c is *univalent* if there exists a cut d containing c such that in d some process has given output but c is not multivalent. A univalent cut is said to have *projected value* 0 if it is contained in a cut in which the empty set is given as output; otherwise, it is said to have *projected value* 1. As a binary relation on outputs, weak collective consistency is not an equivalence relation; but its projection onto values $\{0, 1\}$ is an equivalence relation. A cut c is *nullvalent* if it is not contained in any cut that has an output event. Note that multivalence, univalence, and nullvalence form a strict trichotomy.

A process p *decides* c if c is multivalent and for all d containing c such that p acts in $d - c$, d is univalent. A cut is *undecided* if it is either multivalent or nullvalent. A process p is *ready* at a cut c if there exists an event e_p at p and a cut d such that $d = c \cup \{e_p\}$. A process p is *permanently blocked* at c if there is no cut d containing c such that p acts in $d - c$. Note that if p is permanently blocked at a multivalent cut c then p trivially decides c.

Our results apply to any system satisfying the axioms of *compatibility of independent actions* and *separability of events*, described below. Note that separability of events is equivalent to there being no cycles in the causal relation between events.

- **Compatibility of Independent Actions:** Let c be a cut, and let d_p and d_q be cuts containing c such that $d_p - c = \{e_p\}$ and $d_q - c = \{e_q\}$, where e_p (respectively, e_q) is a single event occurring at process p (respectively, q). The compatibility axiom states that if $p \neq q$ then $d_p \cup d_q$ is a cut.
- **Separability of Events:** Let e_1 and e_2 be two distinct events contained in a cut c. The separability axiom states that there is a cut d containing one of these events but not the other. That is, there exists a d such that $e_1 \in d \wedge e_2 \notin d$ or $e_2 \in d \wedge e_1 \notin d$.

5.2 The Lower Bound

Lemma 2. *For all cuts c, at most one process both decides and is ready at c.*

Proof. Assume for the sake of contradiction that the lemma is false; specifically, let $p \neq q$ both be ready at and decide a cut c. See Figure 3 for an illustration throughout the proof.

Since p decides c, c must be multivalent. By definition of ready, there are cuts $d_q = c \cup \{e_q\}$ and $d_p = c \cup \{e_p\}$ where e_p (respectively, e_q) is a single event occurring at process p (respectively, process q). Recall that each univalent cut has a unique projected value from $\{0, 1\}$. Because both p and q decide c, d_q and d_p must be univalent with projected values v_q and v_p respectively. By the

Compatibility Axiom, $d = d_p \cup d_q$ is a cut. Cut d is univalent because p decides c, d contains c, and p acts in $d - c$. If d had a projected value different from v_p, then d_p would be multivalent. If d had a projected value different from v_q, then d_q would be multivalent. Thus $v_p = v_q$. Let v represent this projected value.

Since c is multivalent, there must exist a cut c' containing c in which some process has given an output with projected value $v' \neq v$. By the Separability Axiom there is a chain of cuts $c = c_0, c_1, c_2, \ldots, c_m = c'$ where each $c_i = c_{i-1} \cup \{e_i\}$, for $i > 0$, and each e_i is a single event. Note that since $c' = c_m$ has an output with projected value v', if any cut c_i is univalent then it is univalent with projected value v'. Also, note that $c_m \cup \{e_q\}$ is not a cut. This is because $d_q = c_0 \cup \{e_q\}$ is univalent with projected value $v \neq v'$ and $c_m \cup \{e_q\}$ contains both d_q and c_0. Thus, there is a least $i \geq 1$ such that $c_{i-1} \cup \{e_q\}$ is a cut but $c_i \cup \{e_q\}$ is not. Fix this i.

By application of the Compatibility Axiom to cut c_{i-1} and events e_q and e_i, event e_i must occur at process q.

Cut c_{i-1} is multivalent because cut $c_{i-1} \cup \{e_q\}$ is univalent with projected value v and cut $c_m = c_{i-1} \cup_{j=i}^{m} \{e_j\}$ has some process giving output with projected value $v' \neq v$. If either p or q acted in $c_{i-1} - c$, then c_{i-1} would be univalent because both p and q decide c. Thus neither p nor q acts in $c_{i-1} - c$.

Since q decides c and q acts in $c_i - c$, cut c_i is univalent. Since c_m has some process giving output with projected value v' and c_m contains c_i, c_i must be univalent with projected value v'. Thus $d_p \cup c_i$ cannot be a cut because d_p is univalent with projected value $v \neq v'$. But, by $i - 1$ applications of the Compatibility Axiom, using the fact that p does not act in $c_{i-1} - c$, $d_p \cup c_{i-1}$ is a cut. Now by the Compatibility Axiom, e_i must occur at process p.

We now have the desired contradiction: event e_i must occur at both process p and process q, contradicting the assumption that $p \neq q$. This completes the proof of the lemma.

Recall that a cut is undecided if it is either multivalent or nullvalent. The proofs of the following two lemmas are immediate from the definitions.

Lemma 3. *If c is multivalent and p does not decide c, then there exists a cut d containing c such that p acts in $d - c$ and d is undecided.*

Lemma 4. *If c is nullvalent and p is not permanently blocked at c, then there exists a cut d containing c such that p acts in $d - c$ and d is undecided.*

Lemma 5. *If cut c is undecided, then there is an undecided cut d containing c such that, for all processes p,*

- *p decides d, or*
- *p is permanently blocked at d, or*
- *p acts in $d - c$.*

Proof. If p is permanently blocked at a cut x, then p is permanently blocked at every cut containing x. If p decides a cut x and an undecided cut y contains x,

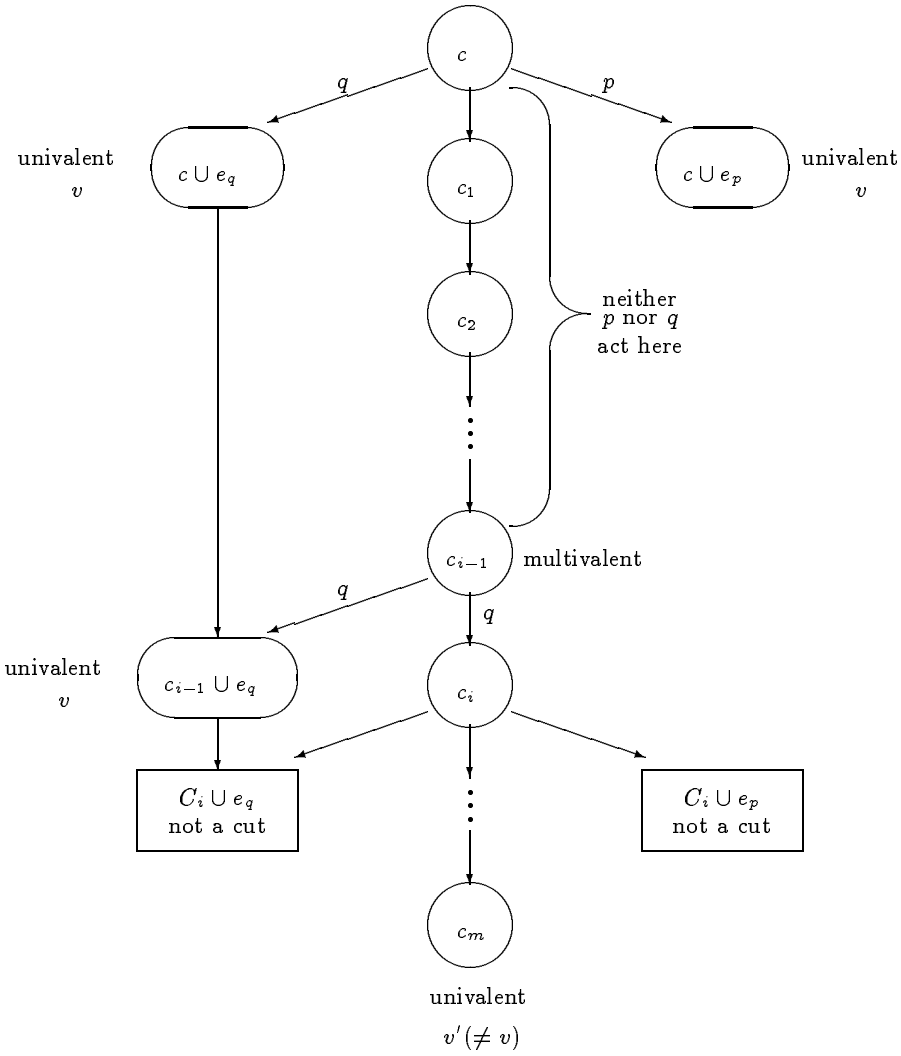

Fig. 3. An illustration of the argument for Lemma 2.

then either y is multivalent and p decides y or y is nullvalent and p is permanently blocked at y. Thus the lemma follows from repeated application of either Lemma 3 or Lemma 4, once for each process.

We now prove that for every nontrivial protocol there is a set W, either empty or containing exactly one process, such that, roughly speaking, (1) if W is empty then the protocol has an execution in which no process fails but no

process ever gives output; and (2) if $W = \{p\}$ for some p, then in some execution every process waits for p. Note that the proof is not restricted to protocols for collective consistency, but applies to *any* nontrivial protocol as defined above.

Theorem 6. *For any nontrivial protocol $\mathcal{P}$, there is a set W containing at most one process such that, for all $n \geq 0$, there exists an undecided cut c such that for all $q \notin W$ either q is not ready at c, or in c at least n events have occurred at q. Moreover, these undecided cuts form a containment chain.*

Proof. The proof is by induction on n. Since $\mathcal{P}$ is nontrivial the empty cut is multivalent. Let c_0 be the empty cut (or any undecided cut). Let $D(c)$ be the set of processes that decide or are permanently blocked at cut c. By lemma 2 at most one process in $D(c_0)$ is ready at c_0. In general, let c_i be an undecided cut such that every process not in $D(c_i)$ has had at least i events. By lemma 5 there is a cut c_{i+1} such that every process not in $D(c_{i+1})$ has had at least $i+1$ events. Thus by induction there is a cut c_n such that every process not in $D(c_n)$ has had at least n events. By lemma 2 at most one process in $D(c_n)$ is ready at c_n. Moreover, once a process is ready and decides a cut, it remains ready and decides every containing undecided cut. Thus if p is ready at and decides c_i then p will be ready at and will decide c_{i+1}. So there is at most one process in $W = \{p | (\exists i) p \in D(c_i)$ and p ready at $c_i\}$.

6 Discussion

In the context of n processes tolerating $c < n$ crash failures and protocols that take input and return output, we call a protocol *non-blocking* if (1) there is a uniform bound on the size (number of protocol relevant events) of a cut in which no process has given output for all executions with at most c crash failures and (2) at each cut of such an execution in which no process has given output, there is at least one process that is not permanently blocked. Our lower bound result shows that if a nontrivial protocol has the property that no undecided cuts contain output events, then it cannot be nonblocking. Since any protocol that achieves at least weak collective consistency has the property that an undecided cut cannot contain an output event, there is no non-blocking protocol that solves weak collective consistency or any stronger version of the problem. In light of this result, we offer the forms of figure 1 with $B = 1$ using either Ψ_1 or Ψ_2 as reasonable choices for protocols that solve weak monotone collective consistency.

As discussed in Section 4, we conjecture that there is a tradeoff between more complicated collective consistency protocols that permit more processes to return and keeping short the window of vulnerability to failure or sudden slowdown, and we have described our intuition for this conjecture. The next logical research step is to design simlation studies to evaluate this conjecture. We believe this tradeoff between protocol sophistication (requiring more rounds of communication) and vulnerability to failures (more rounds during which failures can occur) is not limited to collective consistency protocols; thus any insight gained in learning

how to measure protocol effectiveness in the presence of such a tradeoff will have wide applicability.

The open ended nature of the MPI makes it possible to implement our proposed failure detection and collective consistency protocols without any change to the standard. However, our proposal certainly requires major additions and possible revisions to any communication package that supports the MPI in order to provide our fault tolerance services. Although many readers of an early version of this paper were shocked to learn that timeouts were not an option on MPI calls, we know of no such communication package that is currently available and provides any kind of failure detection by timeout.

Acknowledgements

The authors would like to thank Joe Halpern and Yoram Moses for helpful comments, particularly on the material on knowledge based specifications. They also thank Marc Snir for pointing out that our proposed failure detection and collective consistency services could be offered in the context of the current MPI standard without requiring interface modification.

References

[AW] R. J. Anderson and H. Woll, Wait-free Parallel Algorithms for the Union-Find Problem, Proc. 23rd Annual ACM STOC, pages 370-380, 1991.

[BR] J. Buss and P. Ragde, Certified Write-All on a Strongly Asynchronous PRAM, Manuscript, 1990.

[BW] M. F. Bridgeland and R. J. Watro, Fault-Tolerant Decision Making in Totally Asynchronous Distributed Systems, Proc. 6th Ann. ACM Symp. on PODC, pages 52-63, 1987.

[CHT] T. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, Proc. 11th ACM Symp. on PODC, pages 147-158, 1992.

[CHT2] T. Chandra, V. Hadzilacos, S. Toueg, On the Impossibility of Group Membership, Proc. 15th ACM Symp. on PODC, 1996.

[CT] T. Chandra and S. Toueg, Unreliable failure detectors for asynchronous systems, Proc. 10th Ann. ACM Symp. on PODC, pages 325-340, 1991, to appear in *J. ACM*.

[DMY] Roberto De Prisco, Alain Mayer, Moti Yung, Time-Optimal Message-Efficient Work Performance in the Presence of Faults, Proc. 13th ACM Symp. on PODC, pages 161-172, Los Angeles, 1994.

[DDS] D. Dolev, C. Dwork, L. Stockmeyer, On the minimal synchronism needed for distributed consensus, *J. ACM* 34:1, pages 77-97, 1987.

[DMS] D. Dolev, D. Malki, R. Strong, A framework for partitionable membership service, Proc. 15th ACM Symp. on PODC, 1996.

[DHS] C. Dwork, J. Halpern, and R. Strong, Fault Tolerant Load Management, application for patent.

[DHW] C. Dwork, J. Halpern, and O. Waarts Accomplishing Work in the Presence of Failures. Proc. 11th Ann. ACM Symp. on PODC, pages 91-102, 1992.

[DLS] C. Dwork, N. Lynch, and L. Stockmeyer, Consensus in the presence of Partial Synchrony, *JACM 35*(2), 1988.

[FHMV] R. Fagin, J. Halpern, Y. Moses, M. Vardi, Knowledge-Based Programs, Proc. 14th Ann. ACM Symp. on PODC, pages 153-163, 1995.

[FLP] M. Fischer, N. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process. *J. ACM* 32:2, pages 374-382, 1985.

[Gif] D. K. Gifford, Weighted Voting for Replicated Data, Proc. 7th SOSP, pages 150-159, 1979.

[GMY] Zvi Galil, Alain Mayer, Moti Yung, Resolving Message Complexity of Byzantine Agreement and Beyond, private communication, 1995.

[KPRS] Z. Kedem, K. Palem, A. Raghunathan, and P. G. Spirakis. Combining Tentative and Definite Executions for Very Fast Dependable Parallel Computing. Proc. 23rd ACM STOC, pages 381-389, 1991.

[KPS] Z. M. Kedem, K. V. Palem, and P. G. Spirakis, Efficient Robust Parallel Computations. Proc. of 22nd ACM STOC, pages 138-148, 1990.

[KS] P. Kanellakis and A. Shvartsman, Efficient Parallel Algorithms Can Be Made Robust, Proc. 8th Ann. ACM Symp. on PODC, pages 211-219, 1989.

[KS2] P. Kanellakis and A. Shvartsman, Efficient Parallel Algorithms on Restartable Fail-Stop Processes, Proc. 10th ACM Symp. on PODC, pages 23-25, 1991.

[LA] M. Loui and H. Abu-Amara, Memory requirements for agreement among unreliable asynchronous processors, in F. Preparata ed., *Adv. in Computing Research* 4, pages 163-183, JAI Press, 1987.

[MSP] C. Martel, R. Subramonian, and A. Park, Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs. Proc. 32nd IEEE Symp. on FOCS, pages 590-599, 1991.

[NW] M. Naor and A. Wool, The load, capacity, and availability of quorum systems, Proc. 35th IEEE Symp. on FOCS, pages 214-225, 1994.

[PW] David Peleg and Avishai Wool, Crumbling Walls: A Class of Practical and Efficient Quorum Systems (Extended Abstract), Proc. 14th ACM Symp. on PODC, pages 120-129, 1995.

[R] M. Rabin, The Choice Coordination Problem, *Acta Informatica* 17, pages 121-134, 1982.

[RB] A. Ricciardi and K. Birman, Using process groups to implement failure detection in asynchronous environments, Proc. 10th ACM Symp. on PODC, pages 341-352, 1991.

[W] Arliss Whiteside, Morris Freedman, Omur Tasar, Alexander Rothschild, Operations Controller for a Fault-Tolerant Multiple Computer System, U.S. Patent 4,323,966, 1982.

This article was processed using the LaTeX macro package with LLNCS style