ON THE RELATIONSHIP BETWEEN THE ATOMIC COMMITMENT AND CONSENSUS PROBLEMS

Vassos Hadzilacos

Department of Computer Science
and Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

1. INTRODUCTION

A critical task in the execution of a transaction in a distributed database system is to ensure its consistent termination. The sites whose databases were updated by the transaction must agree on whether to commit it (in which case the transaction's updates will take effect at all these sites) or abort it (in which case none the transaction's updates will take effect at any site). The consistency of the decisions (commit or abort) reached at different sites must be safeguarded even in the presence of various types of failures. This, in a nutshell, is the *atomic commitment problem.* A number of protocols have been devised to solve this problem including, notably, the two-phase commit protocol (Lampson and Sturgis [1976], Gray [1978]).

From this sketchy description it is apparent that atomic commitment is a problem that involves the achievement of some sort of agreement in a fault tolerant manner. Distributed consensus in its various formulations, the focus of voluminous investigation in recent years, is another problem that entails fault tolerant agreement among processes.

The purpose of this paper is two-fold. First, we offer a precise specification for the atomic commitment *problem.* Note the emphasis on the word "problem". We wish to specify, not some protocol that solves the problem, but the problem itself. The specification of the atomic commitment problem is given in the form commonly employed to specify the consensus problem and its variants, namely as a list of conditions that must be satisfied. This facilitates the second purpose of the paper, which is to investigate the relationship between the atomic commitment and consensus problems. We find that the problems, though closely related, are not equivalent; and that the theoretical work on the consensus problem has some interesting and important implications regarding atomic commitment.

The specification of atomic commitment and the study of its relationship to consensus are presented in the two remaining sections of the paper.

2. SPECIFICATION OF ATOMIC COMMITMENT

We assume the following model of computation. There is a collection of processes capable of communicating with each other by exchanging messages. The behaviour of each process is specified by a programme. In effect, this programme stipulates how the process should change its local state, what messages to send to other processes, and whether to accept messages from other processes.

At any time a process is either *up* or *down*. While it is up, the process is following exactly the actions specified by its programme. While it is down, the process takes no actions at all. In the transition from being up to going down, the process' local state is destroyed except for a part, called the *stable state*.† The transition from up to down can be effected at any time. When this transition takes place, we say that a *process failure* occurred. The transition from down to up can also be taken at any time, but after coming up the process must first execute a special programme, called the *recovery procedure*, before it does anything else. The purpose of this procedure is to construct a state from which the process can continue executing its programme. This state must be constructed on the basis of the stable state, which is the only information available to the process when it recovers. When the transition from down to up takes place, we say that the process failure has been *repaired*. A process is *correct* if it has *never* been down; otherwise, it is *faulty*.‡ (Notice that, by this definition, a process may be both faulty and up.)

In addition to process failures, we also admit the possibility of communication failures. When process p expects a message from process q, it sets a period of time, called the *timeout period*, which limits how long p will wait for the message. We say that a process p *cannot communicate with* process q if p does not receive a message it is expecting from q within the timeout period, as measured on p's clock. A *communication failure* occurs when two processes are both up and one is unable to communicate with the other. Note the caveat "are both up": the inability of p to communicate with q because p or q is down counts as a process, *not* communication, failure.

Communication failures may be attributed to a variety of causes: network partitions, unanticipated message delays, fast clocks measuring timeout periods, etc. We do not care what the cause is; what characterises a communication failure is the effect, namely the inability of processes that are up to communicate. We say that a communication failure is repaired if at some earlier time p could

† In practical terms, the stable state consists of the information saved by the process in non-volatile storage (e.g. disk).

‡ Given this definition of a "correct" process, one might be led to conclude that if processes are subject to failure with non-zero probability, eventually they will all become faulty. This is not true because, for our purposes, processes have limited lifetimes. That is, a process is a particular execution of a programme which, in the absence of failures, terminates.

not communicate with q, even though both were up, but at the present time p can communicate with q.

As was mentioned above, each process measures the timeout period on its own clock. A process sets a timeout period on the basis of certain assumptions such as how fast other processes are and how long it takes messages to traverse communication links. To the extent that these assumptions are valid and the process' clock is not too fast relative to some externally defined "absolute time", the process will succeed in receiving the messages it is waiting for within the timeout period it has set. The fact that processes' clocks must have some relationship to a common measure of time means that these clocks must, in some sense, be "synchronised". We shall not attempt to define "synchrony" in this paper, as this would require a long digression with no direct bearing on the subsequent discussion. As far as our model is concerned, violations of "synchrony" (for any reasonable definition of the term) will be manifested as communication failures. (For a discussion of synchrony in the context of the agreement and consensus problems, see Fischer *et al.* [1985], Dolev *et al.* [1983], Dwork *et al.* [1984].)

Given these assumptions on the model of computation, we can specify the atomic commitment problem as follows. We have a set of processes each of which has one input variable, which we shall call the "vote", and one output variable, which we shall call the "decision".

The vote is initially set to one of two values: **Yes** or **No**. The decision has one of three values: **Undefined**, **Commit** or **Abort**. The decision is a write-once variable and its initial value is **Undefined**. If a process' decision is not **Undefined**, we say that the process has *reached a decision.* We say that a process *decides to* **Commit** (respectively **Abort**) if it sets its decision to **Commit** (respectively **Abort**).

The *atomic commitment (AC) problem* is to devise a protocol (i.e. define programmes for these processes) so that the following conditions are satisfied:

AC1 *No two processes reach different decisions.*

AC2 **Commit** *is decided only if all votes are* **Yes***.*

AC3 *If there are no failures and all votes are* **Yes***, then all processes decide to* **Commit**.

AC4 *If all existing failures are repaired and no new failures occur for a sufficiently long period of time, then all processes will reach a decision.*

Some discussion of these conditions is now in order. First, note that we do not require all processes to decide. This would be an unattainable goal because a process might go down and never recover. There is nothing we can require of such a process since, by definition, a process cannot do anything while it is down. Given that we can't require all processes to decide, a natural idea is to require that

NB *All correct processes reach a decision.*

Unfortunately, in the computational model we have described, NB is also unattainable; therefore, we do not make NB part of the AC specification. More specifically, it can be shown that for *any* AC protocol, communication failures can happen in such a manner that correct processes cannot reach a decision without risking a violation of AC1 relative to processes with which they are unable to communicate. This is a consequence of Gray's "Generals' Paradox" (cf. Gray [1978], Skeen [1982], Halpern and Moses [1986]).

However, if we assume away communication failures from our model of computation so that only process failures may occur, there exist AC protocols that satisfy NB (cf. Skeen [1982], Dolev and Strong [1982], Lamport and Fischer [1982], Dwork and Skeen [1983], Hadzilacos [1984]). An AC protocol that satisfies NB is called a *non-blocking protocol.*

AC4 is the condition that identifies the circumstances under which processes are required to reach a decision. Fortunately, this condition *is* attainable!

AC2 states that the decision to **Commit** must be reached by consensus. Notice that there is no similar requirement for **Abort**. This "asymmetry" between the two possible decision values — which, as we shall see in the next section, is not present in the consensus problem — is a reflection of the difference in their meaning. The decision to **Commit** implies the responsibility to carry out an action — namely, to record a transaction's updates in the database. Thus, for this decision to be reached, *every* process that will be required to carry out a piece of the action must be consulted to ensure that it is, in fact, in a position to carry it out.† The votes are used by processes to indicate their ability (vote = **Yes**) or inability (vote = **No**) to carry out the action. Thus, the decision to **Commit** may be reached only if *all* votes are **Yes**. By contrast, the decision to **Abort** engenders no responsibility of carrying out an action and can, therefore, be reached no matter what the votes are.

Note that we do not require the converse of AC2; it is possible that all votes are **Yes**, yet the decision is to **Abort**. The reason for this becomes clear if we consider a scenario in which a process, say p, goes down before other processes have learned its vote, and all the other process' votes are **Yes**. If we insisted in enforcing the converse of AC2, the other processes would have to wait for the recovery of p before any decision could be reached. Rather than waiting, possibly forever, since p might never recover, it is generally considered much preferable to allow the other processes to decide **Abort**, even though the decision to **Commit** was in principle possible (since p's vote might have been **Yes** as well).

AC3 is a partial converse to AC2. It ensures that there are circumstances under which the decision to **Commit** *must* be reached, thereby excluding "so-

† In practical terms, "being in a position to carry out the action" means that the process supervising the transaction's execution at a site has saved the transaction's updates at that site in its stable state, so that subsequent failures will not result in their loss.

lutions" in which all processes always decide to **Abort**.

3. ATOMIC COMMITMENT AND CONSENSUS COMPARED

We now turn to the question of how the atomic commitment problem, as specified previously, relates to the consensus problem. To facilitate this, let us translate the latter into the terms (votes, decisions etc.) we used to specify AC. The problem is to devise a protocol that will guarantee the following two conditions:

A *[Agreement] All non-faulty processes reach the same decision;* and

V *[Validity] If all non-faulty processes' votes are* **Yes** *then they will all decide to* **Commit***; if all non-faulty processes' votes are* **No** *then they will all decide to* **Abort**.

Two weaker versions of this problem which, in fact, are more relevant to our comparison, have also been studied in the literature. These are obtained by replacing V with either

WV *[Weak Validity] If there are no failures then V holds;* or

VWV *[Very Weak Validity] Both* **Commit** *and* **Abort** *are possible decision values; i.e. there is an execution in which correct processes decide to* **Commit** *and an execution in which correct processes decide to* **Abort**.

Satisfaction of A and V is the *consensus problem.* Satisfaction of A and WV is the *weak consensus problem.* Satisfaction of A and VWV is the *very weak consensus problem.*

The main differences between AC and the various versions of consensus concern (a) the decisions reached by faulty processes; and (b) the strength of the conditions required.

Decisions of faulty processes

In the consensus problems the conditions that must be attained do not mention the votes or decisions of faulty processes. On the other hand, AC1 requires that *all* decisions be consistent — be they reached by correct or by faulty processes. Also, AC2 states that the decision to **Commit** is reached only if the votes of *all* processes, including faulty ones, are **Yes**.

Consequently, it is perfectly reasonable to discuss consensus problems in a model that places no restrictions on the possible behaviour of faulty processes (known as the "arbitrary" or "malicious" or "Byzantine" failure model). By contrast, AC is unattainable in such a model. If failures are unrestricted, a faulty process can set its decision variable to an arbitrary value — in particular, a value different from the decision of other processes, contradicting AC1.

In fact, AC is attainable only under the assumption that process failures are of the benign variety mentioned in the beginning of the previous section. That is, processes can fail by stopping (whereupon they lose their state, except for the stable part) and can later recover. By contrast, in the consensus problems there is no such thing as "process recovery". Once a process has gone down, it has become faulty and its actions are therefore irrelevant as far as the consensus problem specification is concerned.

This difference has its roots at the different computing paradigms that motivate the two problems. Consensus originated from a problem in real-time process control (cf. Pease *et al.* [1980], Wensley *et al.* [1978]). In this scenario process decisions are used to trigger some "real world action", e.g. open a valve, which must be carried out within strict deadlines; there is no time to wait for faulty processes to recover and reach a decision. Faulty processes are therefore ignored in the hope that there will be enough correct processes, so that using their decisions alone, the "real world action" will be correctly carried out.

AC, on the other hand, emerged from the transaction processing paradigm. In this case, reaching inconsistent decisions means that the effects of a transaction will not be consistently recorded at all the sites affected by the transaction. This implies that the database itself would become inconsistent. Inconsistent decisions are unacceptable in this case. On the other hand, in the transaction processing paradigm, there are no critical real-time deadlines to be met. Therefore, delayed decisions by processes when failures occur are (grudgingly) accepted.

Strength of the conditions

The second major difference between AC and consensus concerns the strength of the required conditions. It is not difficult to see that conditions AC2, 3 and 4 imply WV. AC3 says that if there are no failures and all processes' votes are **Yes**, all processes will decide to **Commit**; AC2 and AC4 say that if there are no failures and some process' vote is **No**, all processes will decide to **Abort**. Thus if AC2, 3 and 4 hold, so does WV. It is noteworthy that the converse is not true. For instance, under WV it is legitimate to have correct processes decide to **Commit**, when some processes' votes are **No** and others' are **Yes**, while this is not allowed under AC2.

Now, if AC1 — possibly in combination with AC2-AC4 — implied A, we would have that the AC conditions are stronger than those of weak consensus. Unfortunately, that's not quite so. The reason is that in weak consensus all correct processes are required to reach a decision; while in AC correct processes are not required to reach a decision — unless all failures are repaired and no new failures occur (for sufficiently long).

We'll get rid of this problem by assuming that the proviso under which processes are required to reach a decision in AC will eventually be true in any execution. More precisely, we shall assume that the following axiom holds:

NC [No-Catastrophe Axiom] Eventually, all process or communication failures are repaired and no new failures occur for a period of time δ, for any $\delta > 0$.

For the transaction processing type of applications where AC is used, this is a realistic axiom. In fact, given the parameters of a particular system, such as maximum message delays and accuracy of processes' clocks) and a particular AC protocol, it will be sufficient that NC hold for some particular value of δ.†

It is now immediate that AC1, AC4 and NC imply A: AC4 says that if failures are repaired and no new ones happen for sufficiently long (which NC ensures will eventually be the case) then all processes — and, in particular, all correct processes — will reach a decision; AC1 then says that these decisions will be consistent.

We have therefore shown that the AC conditions are stronger than those of the weak consensus problem, provided that the No-Catastrophe Axiom holds. Consequently,

> *all lower bound results that are known about the weak (or very weak) consensus problem are, a fortiori, applicable to AC.*

This is especially important in view of the fact that several lower bound results have been obtained about these problems.‡

A consensus protocol is necessarily non-blocking, since A implies NB. Like atomic commitment with NB, weak consensus is unattainable in the presence of communication failures. In hindsight, this may well be the reason why communication failures are routinely excluded from consideration in the agreement and consensus literature: otherwise, the problems addressed there become unsolvable.

ACKNOWLEDGMENTS

I thank Barbara Liskov and Barbara Simons for their comments on an earlier draft of this paper.

---

† NC is not reasonable for systems that are either too large or are made up of fairly unreliable components. In such systems it may well be that there is never a time during which there are no failures. In fact, NC is stronger than necessary. We could get by with the following, weaker axiom:

> *[Finite Progress Axiom] Failures cannot forever prevent processes from making progress towards the execution of their programmes. In particular, a process cannot remain down forever, when it recovers it can execute at least one step (that will affect its stable state) before going down again, and a message need only be sent a finite number of times before it is received.*

‡ In fairness, it should be pointed out that not all of these bounds are relevant, since some refer specifically to the arbitrary failure model which is far weaker than the failure model in which AC is attainable (cf. earlier discussion on this matter).

REFERENCES

1. Dolev D., C. Dwork and L. Stockmeyer [1983]. "On the minimal synchronism needed for distributed consensus". *Proc. of the 24th Symp. on Foundations of Computer Science*, pp. 393-402, 1983.
2. Dolev D. and H.R. Strong [1982]. "Distributed commit with bounded waiting". *Proc. of the 2nd Symp. on Reliability in Distributed Software and Database Systems*, pp. 53-60, 1982.
3. Dwork C., N.A. Lynch and L. Stockmeyer [1984]. "Consensus in the presence of partial synchrony". *Proc. of the 3rd Annual ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pp. 103-118, 1984.
4. Dwork C., and D. Skeen [1983]. "The inherent cost of non-blocking commitment". *Proc. of the 2nd Annual ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pp. 1-11, 1983.
5. Fischer M.J., N.A. Lynch and M.S. Paterson [1985]. "Impossibility of distributed consensus with one faulty process". *Journal of the ACM* 32(2):374-382 (1985).
6. Gray J.N. [1978]. "Notes on database operating systems". In R. Bayer, R.M. Graham and G. Seegmuller (eds.), *Operating Systems: An Advanced Course.* Lecture Notes in Computer Science, vol. 60. Springer-Verlag, 1978.
7. Hadzilacos V. [1984]. *Issues of fault tolerance in concurrent computations.* PhD dissertation, Harvard University, 1984.
8. Halpern J.Y. and Y. Moses [1986]. "Knowledge and common knowledge in a distributed environment". IBM Research Report RJ 4421, 1986 (revised).
9. Lamport L. and M.J. Fischer [1982]. "Byzantine generals and transaction commit protocols". Computer Science Laboratory, SRI International, Op. 62, 1982.
10. Lampson B. and H. Sturgis [1976]. "Crash recovery in a distributed data storage system". Unpublished memorandum, Xerox PARC, 1976.
11. Pease M., R. Shostak and L. Lamport [1980]. "Reaching agreement in the presence of faults". *Journal of the ACM*, 27(2):228-234 (1980).
12. Skeen D. [1982]. *Crash recovery in a distributed database system.* PhD dissertation, University of California at Berkeley, 1982.
13. Wensley J.H., L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak and C.B. Weinstock [1978]. "SIFT: The design and analysis of a fault-tolerant computer for aircraft control". *Proc. of the IEEE*, 66(10):1240-1255 (1978).