Lower Bounds on Consensus

Leslie Lamport
Compaq Systems Research Center

Mon 13 Mar 2000 [10:01]

Abstract

We derive lower bounds on the number of messages and the number of message delays required by a nonblocking fault-tolerant consensus algorithm, and we show that variants of the Paxos algorithm achieve those bounds.

1 Introduction

What is the cost of making a decision in a fault-tolerant distributed system? The answer depends on the class of algorithms we allow and how we measure the cost. We consider asynchronous algorithms and their standard cost measures—namely, the number of messages and the number of message delays. (An algorithm requires k message delays if some execution contains a chain of k messages, each of which cannot be sent before receiving the preceding one.)

The classic result of Fischer, Lynch, and Patterson [3] implies that a fault-tolerant algorithm cannot be asynchronous. However, outside the domain of process control, most fault-tolerant distributed algorithms use time only to detect failure. In the absence of failure, they are essentially asynchronous. Since failures should be rare, the cost of failure recovery need only be reasonable; it need not be optimal. The number of message delays and the number of messages, in the absence of failure, are the significant costs.

How long it takes to make a decision depends on precisely what making a decision means. We take it to mean choosing an output to send outside the system. For example, in a banking system, it could mean sending a message that instructs an automated teller machine (ATM) to dispense $200 to a customer. The sending of an output is irrevocable. The system might later send a second output countermanding the first—for example, sending a message that instructs the ATM to ask the customer to return the $200. But, that does not change the first output. We also assume that what output is sent may affect future outputs. Hence, the output must be made known to at least some of the processors implementing the system.

We assume that an individual processor can decide that the system should generate a certain output. We determine the number of message delays until the output can actually be sent and until some set of processors knows what output has been chosen. We also determine the total number of messages required.

Under our assumptions, the problem of making a decision can be abstracted as the following consensus problem: each processor begins with an input value that only it knows, and the processors must eventually agree upon a single output value that equals some processor's input value. Requiring that processors have no knowledge of other processors' input values rules out solutions that "cheat" by assuming *a priori* knowledge of what the output should be.

We consider consensus algorithms implemented with a network of processors that communicate by message passing. The algorithm should tolerate non-Byzantine failures—that is, failures in which processors may "crash" and messages may be lost. Our lower bounds depend only on the possibility of message loss, but a practical algorithm should also tolerate processor failure. We require the algorithm to be nonblocking, meaning that a majority of nonfaulty processors that can all communicate with one another must eventually choose an output value.

Fault-tolerant algorithms are usually implemented by selecting one processor to be the leader, and having it coordinate the decision-making process. If the leader fails, then a new leader is chosen. There seem to be two conflicting popular beliefs about the cost of making a decision. One is that the leader can simply decide and then inform the other processors of its decision with a single message delay [1]. If the leader fails, the other processors make the decision. The other belief is that a three-phase commit protocol *à la* Skeen [6] is needed. It is not exactly clear what that means, but it seems to imply that at least three message delays are required. If making a decision is interpreted as solving the consensus problem, then neither belief is correct.

We show that consensus requires and can be achieved in two message delays. With only three processors, two of the processors can know the outcome after only one message delay; with more processors, two message delays are needed before any processor knows what output has been chosen. For an N-processor system, the time-optimal algorithm requires about

$N(N-2)/2$ messages to inform all processors of the output. By taking about $N/2$ message delays, an output can be chosen and made known to all processors with only about $3N/2$ messages. In both cases, variants of the Paxos algorithm [2, 4, 5] yield optimal algorithms.

2 The Consensus Problem

We now specify the consensus problem more precisely. We assume a network of processors that communicate by sending messages that can be lost. For the lower-bound result, there is no need to appeal to processor failure, although we do allow the possibility that network partition causes all messages to and from some processor to be lost. Like the ordinary Paxos algorithm, its optimal variants can tolerate processor failure and message duplication as well.

We require that safety always be preserved—that is, two processors can never choose different outputs, regardless of how many messages are lost. For liveness, we require that the system eventually produce an output if a majority of the processors can communicate with one another. To allow for more general algorithms—especially when there are an even number of processors—we generalize from a majority of the processors to a *majority set* of processors. We require that, if there is some majority set M such that no message sent from one processor in M to another processor in M is lost, then an output is eventually chosen and all processors in M eventually learn its value. All we assume of the collection of majority sets is that they satisfy the following conditions.

1. Any two majority sets have at least one processor in common.
2. No majority set consists of a single processor.
3. If a set P of processors is not a majority set, then the set of processors not in P is a majority set.

The first assumption is necessary if there is to be a solution, since two disconnected subnetworks cannot be guaranteed to choose the same output value. The remaining two assumptions insure that the algorithm is nonblocking, since they imply that the failure of any one processor cannot prevent the algorithm from making progress. The third assumption also ensures that there are as many majority sets as possible. These three assumptions are satisfied if the majority sets are taken to be all sets containing more than half the processors, together with half the sets containing exactly half the processors (if there are an even number of processors).

We measure cost under the assumption that all messages are delivered. However, the algorithm cannot rely on messages being delivered; it must maintain consistency in the face of arbitrary message loss.

3 The Lower Bounds

In deriving lower bounds, we assume that the algorithm works in rounds, where a round consists of the sending of messages by some set of processors and the receipt of a subset of those messages. Lower bounds on an algorithm with such synchronous rounds obviously apply to asynchronous algorithms.

We first show that at least one round is needed before any processor can know what value is chosen. Without receiving any message, a processor p knows only its own input value, so that is the only value it can choose as output. However, there exists a majority set M not containing p. If the processors in M can communicate with one another, then they must choose a value. If they cannot communicate with p, then they cannot learn p's input value, so they may choose a different value. Hence, p cannot unilaterally choose its own value.

We now show that at least two rounds are needed before all processors know what value is chosen. Suppose the value chosen is the input of processor p. After the first round, p cannot know whether the messages it sent have been received. Therefore, it cannot know if any other processor knows its value. If no other processor received its messages, then its value could obviously not have been chosen. Hence, p cannot know until the second round that its value was chosen.

If there is a majority set consisting of exactly two processors, then we will see that it is possible for one of those processors to know the output after the first round. However, suppose that every majority set contains at least three processors, and that the input of processor p is chosen. After the first round, no processor q can know if any processor other than it and p knows p's input value. A network partition could have left p and q able to communicate only with one another. In that case, there is a majority set not containing p and q that might eventually choose a value—which might not be p's since they could not know p's input value. So, q could not know that p's value is chosen after round 1. Hence, if there is no majority set consisting of two processors, then at least two rounds are needed before any processor knows what value is chosen.

We can refine these arguments to obtain lower bounds on the number of messages, assuming that n processors must learn what value is chosen. Let m be the number of processors in the smallest majority set, let M be a majority set with m elements, and assume that the value chosen is the input value of a processor p in M. Each processor in M must learn p's value, and must learn that every processor in M knows p's value. Otherwise, some processor q in M cannot distinguish the actual scenario from one in which only some proper subset of the processors in M know p's value—a scenario in which some other majority set must choose a value that may be different from p's. The processors outside M must also learn that the value has been chosen, so they need the same information. To accomplish this in two rounds requires p to send $m-1$ messages in round 1, and for each of the other $m-1$ processors in M to send a message to all $n-1$ processors other than itself. This gives a total of $n(m-1)$ messages. The same information can be transmitted with fewer messages as follows. Processor p sends its input to one processor in M, which relays it to another processor in M, and so on until the message reaches the m^{th} processor in M. That processor then sends a message to all $n-1$ processors other than itself. This uses $m+n-2$ messages and m rounds. It can be shown that this is the minimal number of messages, and any algorithm using that few messages requires at least m rounds.

The bounds given in the introduction were obtained by taking n to be the number N of processors and m to equal about $N/2$. To ensure that the system can make future decisions that depend on the chosen output, it is only necessary that all processors in some majority set learn what value is chosen. Any majority set then contains at least one processor that knows the chosen value.

4 The Algorithm

We have shown that any algorithm requires at least two message delays. An algorithm that uses only two messages delays and informs n processors of the outcome requires $n(m-1)$ messages; an algorithm can use as few as $m+n-2$ messages, but then requires m message delays. We now show that all these bounds can be achieved by variants of the Paxos algorithm—more precisely, by variants of the synod protocol that lies at the heart of Paxos. The synod protocol is a consensus algorithm that assumes that a leader has been elected. Safety is guaranteed even if there is no leader or there are multiple leaders. Liveness depends on the existence of a single leader. We now sketch the protocol.

The synod protocol consists of three phases. In the first phase, the leader sends a message to every processor in a majority set M and receives a reply. (In this description, a processor can send a message to itself; in the actual algorithm, such a message is not really sent.) In the second phase, the leader sends another message containing its input to the processors in M. Each processor in M replies with an acknowledgment message.

The leader's input value is committed as the chosen value as soon as all the processors in M have received the phase-2 message. The leader knows that the value has been chosen as soon as it has received all the phase-2 acknowledgments. If the leader and another processor q in M form a two-element majority set, then q knows that the value has been chosen as soon as it receives the phase-2 message. If all majority sets contain more than two processors, then the leader is the first to learn that its value has been chosen. In phase 3, the leader sends messages to the other $n-1$ processors informing them that its input was chosen.

This sketch describes how the synod protocol behaves if there is a single leader and there are no failures or lost messages. The complete synod protocol, which requires saving information in stable storage to protect against processor crashes, can be found elsewhere [2, 4, 5].

The Paxos algorithm for implementing a distributed system uses a sequence of separate instances of the synod protocol. The first phase of all those instances is executed only once. This is possible because the leader does not reveal its input value until the second phase. We turn the synod protocol into consensus algorithms satisfying our lower bounds by starting it in the state following completion of phase 1. To obtain the two-message-delay lower bound, we modify phase 2 by having processors send their acknowledgments to all processors, not just to the leader. A processor knows that the leader's value has been chosen when it receives the acknowledgments from all processors in M other than the leader. This produces a two-message delay algorithm with the optimal number $n(m-1)$ of messages for such an algorithm. We can reduce the number of messages by packaging the leader's message and/or multiple acknowledgments in a single message. The minimal number of messages, $m+n-2$, is obtained with an m-message delay algorithm as indicated above. The leader sends its phase-2 message to one other processor in M, who relays it and his acknowledgment to the next processor in M, and so on; and the last processor in M combines all those acknowledgments into a message that it sends to the remaining $n-1$ processors. The original synod protocol is a compromise between the two optimal versions, using three message delays and $2m+n-3$ messages.

Acknowledgment

Mark Hayden's comments helped us improve the current version.

References

[1] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[2] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the Paxos algorithm. In Marios Mavronicolas and Philippas Tsigas, editors, *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG 97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125, Saarbruken, Germany, 1997. Springer-Verlag.

[3] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[4] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[5] Butler W. Lampson. How to build a highly available system using consensus. In Ozalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, 1996. Springer-Verlag.

[6] Marion Dale Skeen. *Crash Recovery in a Distributed Database System.* PhD thesis, University of California, Berkeley, May 1982.