Reaching Agreement Optimistically in a Faulty Asynchronous System

J. Roger Mitchell* Vijay K. Garg†

Parallel and Distributed Systems Laboratory
http://maple.ece.utexas.edu
Department of Electrical and Computer Engineering
Engineering Science Building
The University of Texas at Austin,
Austin, TX 78712-1084

April 26, 1996

Abstract

Agreement among processes in a faulty distributed system is useful for the solution to leader election, atomic broadcast, and commit protocols, among others. We present a method for solving consensus using optimism. Our approach is optimistic in that processes do not wait on other processes. Instead they assume all proposals have been received and continue execution. Only if this assumption is later proved wrong, are actions based on these tentative decisions required to roll back. With this approach, all processes can decide the same value in a completely asynchronous distributed system that has process, message, or channel failures, and thus can become partitioned. Since no protocol can commit in a distributed system with these characteristics (by the impossibility result), we discuss methods for using optimistic agreement with other commitment algorithms. When employed this way, the approach provides benefits of continued execution in a partition, optimistic execution during commit, and a loose failure model for the asynchronous agreement group.
Keywords: fault tolerance, optimistic, consensus, distributed system.

1 Introduction

In a distributed system the solution to the agreement problem is important for distributed databases and file systems, synchronization problems such as mutual exclusion, and is applicable to leader election and atomic broadcast. There has been a great deal of work in this area, much of which followed the impossibility result of Fischer, Lynch and Paterson [9]. The FLP result demonstrated that any asynchronous commit protocol can wait indefinitely in the presence of a single fail-stop failure. This result does not rely on faulty message delivery, but rather the problem that a failed

*supported in part by MCD and Virginia & Ernest Cockrell fellowships

†supported in part by the NSF Grants ECS-9414780, CCR-9520540, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

process cannot be distinguished from a slow process. Because of this problem, a process does not know whether to wait for an unreceived proposal or to assume that the process has failed. Many researchers have devised methods for working around the impossibility result by limiting the faults or limiting asynchrony [2, 4, 5, 7, 8]. These researchers, however, constrain a process from agreeing until it is sure that the decision is final.

We will present a method called *optimistic agreement* which overcomes the problem of the FLP result by having processes make a tentative decision immediately, with the ability to undo this decision at any time if future information changes the decision. Therefore, a process never waits on a process that may be failed. Our system model is completely asynchronous and allows for unreliable messages and fail-stop processes. Unlike previous work, optimistic agreement allows for processes to continue to make the most progress with the information available in spite of faults or a partitioning of the network.

Most previous work that addresses consensus in a faulty environment circumvents the asynchronous nature of the system and does not allow for progress in the case of partitioned networks. [2, 4, 5, 7, 8] give methods for solving consensus by working around the problem of the impossibility result. This is done using partial synchrony, probabilistic methods, randomization techniques, or a weakening of the problem. The work of [5] uses failure detectors that suspect which processes have failed. Failure detectors usually use timeouts. In general, methods of inserting some synchronous behavior into the environment are employed in previous work.

The problem of replicated database consistency has received a lot of attention. This problem uses consensus to force the changes across replicas to be made consistently. Most work in database consistency has used quorums. This approach requires that to be able to read from the database, r replica responses be obtained, and for a write, w replica responses be obtained. Then, $r + w > n$, where n is the total number of copies (replicas) of the database. In [6], two approaches to consistency in a partitioned database are presented, *pessimistic* and *optimistic*. A pessimistic consistency algorithm assumes that the worst will occur during a network partitioning and therefore is unlikely to commit to a change of the database. Optimistic approaches assume the inconsistency is rare and therefore provide greater availability. In both cases, any inconsistencies found after a network rejoins requires that some transactions be undone. Agrawal and Bernstein [1] present a method for implementing optimistic reads by having a reader assume that the first response is up to date. To do this with a high probability and avoid rolling back, the approach employs a background propogation mechanism for writes which spreads the updates to all replicas. Similar to these, our work in agreement is optimistic; a process makes a decision immediately upon receiving a proposal and continues execution. Only if this proves to be wrong does it reverse its decision.

2 Properties of an agreement protocol

The agreement problem can be stated as: all processes start with an initial value or proposal. After sharing these proposals a non-faulty process chooses a decision value. All other non-faulty processes should choose the same value. The problem can be formalized by stating it as properties. Any protocol that displays these properties can then be considered an agreement protocol. A standard definition of agreement follows:

> A protocol provides *agreement* among processes in a faulty environment if a correct process's decision of v implies that all correct processes will eventually decide v.

Most previous work using the above definition has made the assumption that once a process makes a decision it will not change that decision. Since our approach allows a process to change its decision we add the word *final* to the above definition:

> A protocol provides *agreement* among processes in a faulty environment if a correct process's *final* decision of v implies that all correct processes will have a *final* decision of v.

Here a final decision can be thought of as a decision which does not later change. In addition to satisfying the agreement definition, agreement protocols should also satisfy the following [10]:

- **Termination:** Every correct process decides some value after a finite number of steps.
- **Validity:** If all proposals are equal to v, then all correct processes eventually decide v.
- **Integrity:** No process decides more than one *final* value, and if a process decides a value then it has been proposed.

The validity property is designed to preclude an agreement algorithm which always decides null. In our case, null is not an allowed decision value, therefore integrity implies validity and we will not discuss validity further. In the definition of integrity we have added the italicized text to demonstrate that the meaning of the original definition does not allow commitment to more than one value. For this paper, the added property of knowledge monotonicity will be used to prove the other properties:

- **Knowledge monotonicity:** Knowledge of agreement proposals only increases.

This property ensures that knowledge used to make decisions for agreement is strictly increasing and supports the notion that all such knowledge is eventually shared.

The slight change in the definition of agreement is the difference which allows our algorithm to reach agreement in a completely asynchronous environment. Since the approaches used in other work always make only one decision, a process will do so only when it is sure all other correct processes will make this same decision. However, if tentative decisions are allowed, a process never knows if it will have to later change its decision. The process, then, can never perform an action that cannot be undone. That is, it cannot commit.

Since in optimistic agreement a process never knows if it has reached its final decision, it can never commit. Why then, is optimistic agreement useful? This approach provides the following benefits:

- allows progress in a partitioned network.
- allows the execution to proceed as far as possible before committing.
- allows the agreement group to remain under the assumptions of a faulty, asynchronous environment, when a method for committing is implemented. (Only some outside process needs to employ different assumptions).
- can be used with other agreement protocols, which implement commitment, to provide the benefits listed above.

In the presentation, previous work will be discussed first, followed by an example of our approach, formalization of the algorithm and properties, and then proofs of correctness. We then give complexity of the algorithm, a method for reducing this complexity, and finally how to usefully employ optimistic agreement in an algorith that commits.

3 The method

To facilitate the presentation, optimistic agreement is demonstrated for a single agreement. The algorithm can be easily extended to multiple agreements, by assigning a number to each. With this approach overlapping agreements could be kept separate.

Figure 1 gives an example of the protocol. As shown in the figure the group of processes involved in agreement are $P_1 - P_5$. P_1 has decided to propose a value; it does so by sending a proposal to all members of the group. This occurs at event a in the figure. The proposal to P_2, however, is lost in the channel. Concurrently, P_3 sends its own proposal at b. P_5 receives the proposal from P_1 first and immediately assumes that this is the only proposal it will receive. P_5 sends message m_1 based on P_1's proposal. Message m_1 includes the proposal of P_1, on which it depends. P_3 receives m_1 and rejects it because the message indicates that P_5 did not know about P_3's own proposal. The information about P_1's proposal is retained, however, since P_3 has not yet received this proposal. P_3 generates message m_2 to tell P_5 of its own proposal, since the original proposal may have been lost. (In fact, the original proposal is received before m_2 which is then ignored by P_5.) At this point, P_3 knows of both proposals even though it has not received the original proposal from P_1. P_3 later creates a message, m_3, which depends on these proposals and sends this to P_2. P_2 accepts m_3 because the message contains all the proposals of which P_2 knows (at this point P_2 knows only of P_3's proposal). However, m_3 contains the proposal from P_1 and so P_2 learns of this upon receiving the message.

The approach presented above can be stated as: messages are accepted which contain equal or greater proposal information and rejected which do not contain all known proposals. It is apparent, then, that if all unfailed processes communicate, eventually all these processes will have the same set of proposals and will reach the same decision. It can also be easily seen that the method maintains progress towards agreement, since knowledge about proposals only increases at each process. This will be demonstrated formally in following sections.

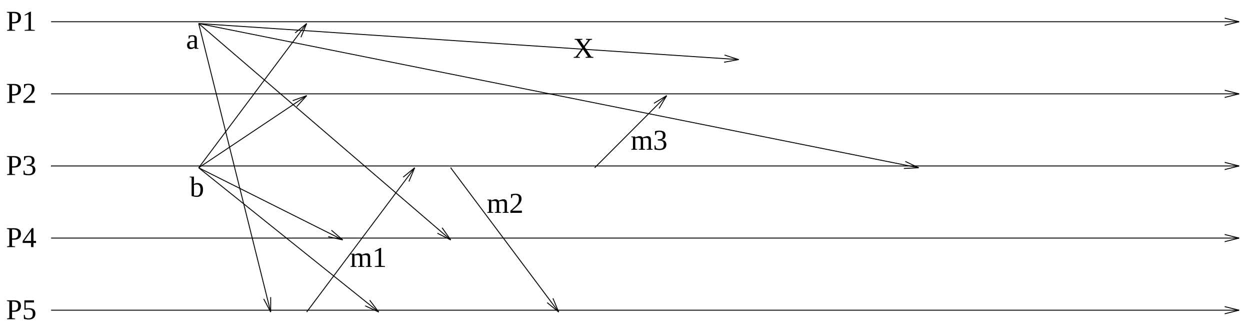

Figure 1: Example operation of optimistic agreement

3.1 Notation and Formalization

Our model is a message-passing system of distributed processes without a global clock or shared memory. Process q in the system is represented by P_q. The executions of processes in the system together make up a run r. The execution of a single process P_q is a sequence of alternating states and actions, $s_{q,1}a_{q,1}s_{q,2}a_{q,2}...a_{q,l-1}s_{q,l}$ where s_q's are states and a_q's are actions. This makes up the run $r[q]$. The ordering of states in our model uses the happened-before relation ($\rightarrow$) on states. We say that a state $t \in r[q]$ happened-before ($\rightarrow$) a state $u \in r[p]$ if and only if 1) $q = p$ and t occurs before u in $r[q]$, 2) the action following t is the send of a message and the action preceding u is the receive of that message, or 3) there is a state v such that $t \rightarrow v$ and $v \rightarrow u$. We say $t \in r[q] \prec u \in r[q]$ if and only if t occurs before u. We say $t \rightsquigarrow u$ if t is the state before the send of a message and u is the state following the receive of that message. Further notation is defined:

- s, t: states at a process.
- m.send: the state at which message m was sent.
- q, r: process index numbers.
- P: the set of all processes.
- C: the set of all unfailed processes.
- P_q: process q.
- S_q: the set of *failure free* states on process P_q.
- $prop_q \equiv$ agreement proposal by process q.
- $s.K \equiv$ the set of proposals known at state s.
- $EK_q \equiv \cup_{s \in S_q} s.K$. We will show later that when $s.K = EK_q$, $s.K$ is stable.
- $s.D \equiv$ decision value at state s. $s.D$ is a function of $s.K$ which we write as $f(s.K)$. Note that $f(s.K) = \phi$ if $s.K = \{\}$.
- dep$(s) \equiv$ The activity performed at state s depends on the value of the agreement. For example, if the value of x computed at state t uses the value of the agreement, then dep(t) is true.

The functions used in $f(s.K)$ for a decision can take the form of majority vote, min/max, logical functions, or others. Any of those listed will have the property that the decision value will be contained in the set of proposal values. This is a requirement for the property of integrity.

Using the above notation, formal definitions of the properties of an agreement protocol can be given.

1. Knowledge is monotonic if:
   $s \rightarrow t \Rightarrow s.K \subseteq t.K$
   Knowledge is monotonic if it does not decrease during the computation.

2. A protocol satisfies *agreement*, if:
   $\exists v, \forall q \in C :: f(EK_q) = v$
   Agreement is satisfied if the final decision of one correct process is the final decision of all correct processes.

3. An agreement protocol will *terminate* if:
   $\forall q \in C, \exists s \in S_q :: s.D = f(EK_q)$
   Agreement terminates if all processes eventually reach a final decision.

4. An agreement protocol has *integrity* if:
   i) $\forall q \in C$ :: There is only one $f(EK_q)$
   ii) $\forall q \in C, s \in S_q :: s.D = v \Rightarrow v \in EK_q$
   Integrity is ensured if all correct processes choose only one value and if the decision value has been proposed.

3.2 Assumptions

Now that the properties of agreement have been established, the assumptions for our model that allow us to satisfy these properties are given.

A1: Messages may fail by not being received.

A2: Processes may fail by crashing: there exists a local state after which all activity ceases.

A3: Processes may fail by becoming isolated: there exists a local state after which the process does not communicate with a majority of other communicating processes either directly or indirectly.

Assumption A3 leads directly to the following two consequences:

- Every unfailed process always eventually communicates with all other unfailed processes. This is written as: $\forall q, r \in C, s \in S_q, \exists t \in S_r : s \rightarrow t$

- Any system partitioning of correct processes does not last forever.

Since a send failure cannot be distinguished from a receive failure A1 can model any type of message loss. Since a channel failure cannot be distinguished from repeated message failures, A1 can also model channel failures. Malicious failures, such as spontaneous message creation or incorrect process behavior, are not allowed. The failure types do allow for a network partition, however. A partition is defined as a group of processes completely isolated from all other processes in the system. However, as mentioned above, any partition separating correct processes will not last forever. Since a failed process never communicates with a correct process, any incorrect tentative decision made by a process before failing does not have an ill effect. In addition, any communication from a process before failing that reaches a process that does not fail, will reach all processes which do not fail. It will be shown that as long as assumptions A1–A3 hold, an asynchronous system with these failure types can reach agreement, as defined earlier.

For state s at process q:

To send a proposal:
Send proposal $prop_q$ to all processes in group using send procedure.

Upon receiving proposal: $(s, prop_r, t)$
$t.K = s.K \oplus prop_r$; /* $\oplus$ = union of proposal information */
if $t.K \neq s.K$ then
reperform any calculations that depend on agreement;

When sending message m: (s,send(m)$, t$)
/* Send procedure. */
include $s.K$ as $m.K$ in message m;
dep(send(m)): include m.dep in message m.

Upon receiving message: (s,receive(m)$, t$)
/* Receive procedure */
if (m.dep in m and $m.K \supseteq s.K$) then
/* accept message m:*/
if ($m.K - s.K \neq \emptyset$) then receive the difference as proposals;
receive application message;
else
/* refuse message m, but accept new agreement information:*/
for all $prop_{m.\text{sender}} \notin s.K$ receive as proposals;
Send null message using send procedure to originator of m.

Figure 2: The optimistic agreement algorithm

3.3 The algorithm

Figure 2 gives the algorithm for optimistic agreement. The notation (s, event, t), specifies a local event and the two local states that precede and follow it, following the definition of a run. In the algorithm, whenever the program indicates any type of message is sent, the message is sent using the sending procedure, and whenever any message is received it is done so using the receive procedure.

If a proposal is received it is added to the list for that agreement and any calculations previously performed dependent upon this agreement are redone. We do not show the mechanism for doing this in the algorithm since these are given in the literature [12].

To enable a process to determine whether to accept a message, all messages sent indicate the proposals on which they depend. To accept a message, the proposals on which the message depends must be a superset of those at the destination. If the destination accepts the message then all new proposals are also received. If the destination does not accept the message, then any proposals known by P_q which are not contained in the message are forwarded to the sender.

4 Properties of our algorithm

In this section, we demonstrate that our algorithm has the properties of an agreement algorithm. Recall that for our system model, no assumption has been made about message ordering or timing, that is, the model is completely asynchronous.

Theorem 4.1 *Optimistic agreement provides knowledge monotonicity.* $(s \rightarrow t \Rightarrow s.K \subseteq t.K)$

Proof:

$s \rightarrow t$

$\Rightarrow$ There is a chain from s to t such that for all $s_k \in$ chain: $s_k \prec s_{k+1} \vee s_k \rightsquigarrow s_{k+1}$
{definition of $\rightarrow$}

$\Rightarrow$ $s.K \subseteq t.K \vee s.K \subseteq t.K$
{for $\prec$, t will have greater knowledge than s only if there is an intervening proposal event for $s \prec t$ and because of failure assumptions A1, A2, and A3. for $\rightsquigarrow$, by receive proposal and receive message procedures, it is apparent that knowledge only increases.}

$\Rightarrow$ $s.K \subseteq t.K$
{by simplification}

□

Since knowledge only increases, when $s.K = EK_q$, $s.K$ must be stable.

Theorem 4.2 *Optimistic agreement satisfies agreement.*$(\exists v, \forall q \in C :: f(EK_q) = v)$

Proof:

Because every process can only make one proposal eventually some, process will stop receiving new proposals:

$\exists r \in C, u \in S_r : u.K = EK_r$

$\Rightarrow$ $\forall q \in C, u \in S_q : u.K = EK_r$
{Since all correct processes eventually communicate (by A3), knowledge monotonicity, and the fact that EK_r doesn't change.}

$\Rightarrow$ $\forall q \in C, \exists u \in S_q \forall t \prec u : u.K = s.K = EK_r$
{since EK_r is not changing, there cannot be further proposals to receive.}

$\Rightarrow$ $\forall q \in C :: EK_q = EK_r$
{by definition of EK_q.}

$\Rightarrow$ $\exists v, \forall q \in C :: f(EK_q) = v$
{since EK_q is static for all q and are equal for all q.}

□

Theorem 4.3 *Optimistic agreement satisfies termination.*$(\forall q \in C, \exists s \in S_q :: s.D = f(EK_q))$

Proof:

Because agreement must have at least one proposal and from agreement proof, there is a EK_r such that:

$$
\begin{array}{ll}
& \forall q \in C, \exists u \in S_q, \forall t \prec u : u.K = s.K = EK_r \\
\Rightarrow & \forall q \in C, \exists u \in S_q, \forall t \prec u : u.D = s.D = f(EK_r) \\
& \{\text{by definition of } s.D.\} \\
\Rightarrow & \forall q \in C, \exists u \in S_q, \forall t \prec u : u.D = s.D = f(EK_q) \\
& \{\text{from agreement proof, } EK_q = EK_r \text{ for all } q.\} \\
\Rightarrow & \forall q \in C, \exists s \in S_q :: s.D = f(EK_q)) \\
& \{\text{follows directly.}\}
\end{array}
$$

□

Theorem 4.4 *Optimistic agreement satisfies integrity.*
i) $\forall q \in C$ :: *There is only one* $f(EK_q)$
ii) $\forall q \in C, s \in S_q :: s.D = v \Rightarrow v \in EK_q$

Proof:
1) A process reaches one set EK_q. By definition, the function $f()$ returns only one value. This gives result.
2) Since $\exists r \in C, s \in S_r : s.K = EK_r$ (from agreement proof), eventually every correct process will have such a state s. By definition, $s.D = f(s.K)$. Also, $f(s.K) \in s.K$. Therefore, $s.D = f(EK_q)$, which implies the result. □

5 Complexity

In our algorithm, if N is the number of processes trying to reach agreement, it is easy to see that the message complexity is $O(N^2)$ (all processes could make a proposal and therefore send a proposal message to all other processes). This number is similar to other agreement protocols. Michael Ben-Or's probabilistic algorithm [2] requires $O(N^2)$ messages. Chandra and Toueg in their work with eventually weak failure detectors [4] give an algorithm which is $O(fN)$ (where f is the number of processes which can fail or become suspected). Partially synchronous algorithms presented by Dwork, et. al. in [8] proceed in rounds with message complexity of $O(fN)$ as well.

In optimistic agreement, message size is related to the number of proposals. The added proposal information in an application message for any agreement is $O(N)$. If the algorithm is used for multiple agreements, this grows to $O(kN)$ where k is the number of agreements. We will demonstrate that this can be reduced to $O(1)$ in the next section.

5.1 Reducing message size

As the algorithm progresses, each process must include in messages to other processes any proposal known (since the other process may not know about it). The easiest way to do this would be to always send all proposals ever received. Obviously, messages will soon become heavily laden with proposals, and the added size as mentioned in the previous section would be $O(kN)$ for k agreements. This message size can be reduced by maintaining additional information locally. This additional information concerns what other processes know. By keeping track of what other processes know, a process only needs to send new proposal information with messages. Because eventually every correct process will communicate, eventually there will be no need to include old proposal information with messages.

To implement this method, each process maintains the following sets for every other process:

- New_q – the set of proposals for agreement i that a process P_r knows, but is unsure whether P_q knows.

- Old_q – the set of proposals for agreement i that a process P_r knows P_q knows, but is unsure whether P_q knows P_r knows.

Whenever process P_r sends and receives, it adheres to these rules:

- If a proposal $prop_p$ is received, for all q add $prop_p$ to New_q.

- When sending a message to P_q, include New_q and Old_q as $NewM$ and $OldM$.

- When receiving a message from P_q:

  1. if $NewM \neq \{\}$ then add $NewM$ to Old_q and all New_x, where $x \neq q$. Also remove $NewM$ from New_q (if present).

  2. if $OldM \neq \{\}$ then remove $OldM$ from both New_q and Old_q.
     item if $NewM = OldM = \{\}$ then set $Old_q = \{\}$.

Figure 3 demonstrates how two processes upon exchanging messages for the application, soon come to know what the other knows. As in this example, eventually all correct P_q will know of the same proposals, and therefore all New_q and Old_q will be empty. After this point, there will be no proposals exchanged between correct processes. However, for failed or unreachable processes, the sets will grow. If a process could commit to a decision, it could flush the New_q and Old_q for all P_q (including those that are failed). It would then be able to return a message indicating this commitment to any process P_r from which a message has been received which indicates P_r has not committed.

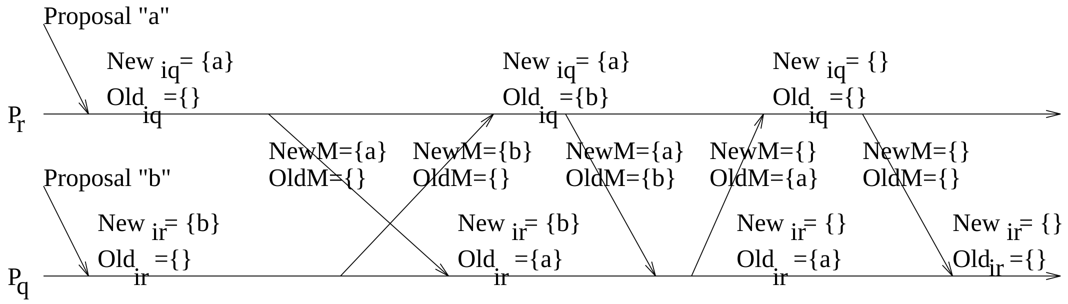

Figure 3: An example of how two processes come to know what each other knows.

In the next section methods are given for reaching commitment in optimistic agreement.

6 Commitment in Optimistic Agreement

In the algorithm presented, we are able to reach agreement among all processes and meet our definitions of an agreement algorithm. It is interesting to note that although the algorithm satisfies the properties of an agreement algorithm it cannot commit. Although a process may know that it will eventually agree with all other processes, it does not know when that will be. All processes must forever assume that sometime in the future a new proposal will reach them that will alter their decision. As long as an irretrievable message does not need to be sent outside of the agreement processes, the algorithm can continue optimistically. However, the group cannot commit.

As soon as a message must be sent outside of the group which is irretrievable, the group must commit. It is apparent that commitment (sending an irretrievable message) can be avoided for awhile because of the optimism of the algorithm: progress will still occur. Note that certain types of agreement, such as leader election, and atomic broadcast, lend themselves to this approach. This is because any process outside the group does not need to know which leader was elected or the actual order of messages. Only when an irretrievable message must be sent that indirectly depends on a decision must the decision be committed. By the impossibility result, this requires the introduction of some synchrony into the environment or a different assumption about failures.

We suggest one solution to the problem. Rather than change the assumptions about the processes attempting agreement, we make assumptions about the outside process to which any irretrievable message is sent. We require this process to either not fail or recover if it does during its attempts to get the group to commit. Alternatively, if it does, it must recover with knowledge of its attempted commitment. The following help justify the use of a reliable outside process:

- If the destination process of an irretrievable message fails before receipt, there is no need to send the message. Therefore, only if the destination process is operating need the message be sent.
- The outside process determines, directly or indirectly, that an agreement needs to be made. Therefore, it can be assumed that the outside process can in some way continue requesting a response until one is received. If the process stops requesting and a response is never received, it has likely failed, in which case a response is not necessary anyway. So, optimistic agreement will continue without committing.

With these assumptions, it is easy to implement commitment. One of several approaches, such as two phase commit or other methods of agreement, can be used. In most cases, the number of processes which can fail in the agreement group must be constrained to a minority. We suggest that optimistic agreement used in this way provides the following benefits:

- allows progress in a partitioned network (when commitment may be temporarily impossible).
- allows the execution to proceed as far as possible before committing (a commitment method can proceed in the background without holding up the optimistic execution).
- allows the agreement group to remain a faulty, asynchronous environment (as described above).

7 Applications of optimistic agreement

As mentioned in the previous section, two agreement problems which might work well with our algorithm are leader election and atomic broadcast.

Suppose leader election were necessary to choose a primary in a group of replicas. To perform the election using optimistic agreement, proposals would be the current id of each processor, and therefore for all P_q, $prop_q$ would be q. The known proposals, $s.K$, would be every process that P_q has heard from, if $s \in S_q$. Finally, $s.D = \max\{s.K\}$, or some other function. When running the leader election, since optimistic agreement is optimistic, every process will begin assuming that the first proposal it receives is the leader. Alternatively, the algorithm could be implemented to wait some amount of local time following the receipt of a proposal to see if another might be received. It will then follow any activity directed by the leader on which it has decided, backing up when it finds an error in its assumption. With either method, eventually all processes will reach the same decision and follow the elected leader until a new leader is required. As soon as a message must be sent outside the group that depends on an elected leader, the election must be committed. This can be done using a method discussed in the previous section.

Atomic broadcast requires that all processes agree on a receive order for a set of messages. In optimistic agreement, proposals would indicate a vote on message order. Any execution that relies on this order must be able to roll back or must commit.

8 Conclusions

Agreement among processes in a faulty distributed system is useful for the solution to leader election, atomic broadcast, and other distributed problems. The method presented here allows for reaching consensus among a group of completely asynchronous processes, in which processes may crash, messages or channels may fail, and the group may become partitioned. Our approach is optimistic in that processes do not wait on other processes. Instead they assume all proposals have been received and continue execution. Only if this assumption is later proved wrong, are actions based on these tentative decisions required to roll back. Previous work in agreement forces processes to wait for a set number of proposals before acting. Our protocol can commit by making assumptions about an outside process to which an irretrievable message is to be sent. In addition, the approach can provide useful benefits when employed with other commitment algorithms. We have also given a method for reducing the amount of proposal information sent with messages to $O(1)$.

References

[1] D. Agrawal, A. Bernstein, "A Nonblocking Quorum Consensus Protocol for Replicated Data," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 2, No. 2, April 1991, pp. 171-179.

[2] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract)," *Proceedings of the 2nd ACM Annual Symposium on Principles of Distributed Computing*, Montreal, Quebec, 1983, pp. 27-30.

[3] E. Borowsky, E. Gafni, "Generalized FLP Impossibility Result for t-resilient Asynchronous Computations (extended abstract)," *Proceedings of the 25th ACM Annual STOC*, California, 1993, pp. 91-100.

[4] T. Chandra, S. Toueg, "Unreliable Failure Detectors for Asynchronous Systems (Preliminary Version)," *Proceedings of the 10th ACM Annual Symposium on Principles of Distributed Computing*, August, 1991, pp. 325-340.

[5] T. Chandra, V. Hadzilacos, S. Toueg, "The Weakest Failure Detector for Solving Consensus," *Proceedings of the 11th ACM Annual Symposium on Principles of Distributed Computing*, Vancouver, B.C., 1992, pp. 147-158.

[6] S. Davidson, H. Garcia-Molina, D. Skeen, "Consistency in Partitioned Networks," *ACM Computing Surveys*, vol.17, no. 3, ACM Press, 1985, pp. 341-370.

[7] D. Dolev, C. Dwork, L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *Journal of the Association for Computing Machinery*, Vol. 34, No. 1, Jan 1987, pp.77-97.

[8] C. Dwork, N. Lynch, L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, vol. 35, no. 2, 1988, pp. 288-323.

[9] M. Fischer, N. Lynch, M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no.2, 1985, pp.374-382.

[10] V. Hadzilacos, S. Toueg, "Fault-Tolerant Broadcasts and Related Problems," *Distributed Systems*, ACM Press, Addison-Wesley, 1993, pp. 97-146.

[11] J. Halpern, Y. Moses, "Knowledge and common knowledge in a distributed environment," *Proceedings of the Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada, August 1984, pp. 50-61.

[12] D. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July, 1985, pp. 404-425.

[13] S. Moran, Y. Wolfstahl, "Extended Impossibilty Results For Asynchronous Complete Networks," *Information Processing Letters*, Vol. 26, 1987, pp. 145-151.

[14] P. Panangaden, K. Taylor, "Concurrent common knowledge: a new definition of agreement for asynchronous systems," *Proceedings of the 8th Symposium on Principles of Distributed Computing*, 1988, pp. 197-209.

[15] J. Turek, D. Shasha, "The Many Faces of Consensus in Distributed Systems," *Computer,* June, 1992, pp. 8-17.