Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors

C. Mohan, R. Strong, S. Finkelstein

IBM San Jose Research Laboratory
San Jose, CA 95193

ABSTRACT: This paper describes an application of Byzantine Agreement [DoSt82a, DoSt82c, LyFF82] to distributed transaction commit. We replace the second phase of one of the commit algorithms of [MoLi83] with Byzantine Agreement, providing certain trade-offs and advantages at the time of commit and providing speed advantages at the time of recovery from failure. The present work differs from that presented in [DoSt82b] by increasing the scope (handling a general tree of processes, and multi-cluster transactions) and by providing an explicit set of recovery algorithms. We also provide a model for classifying failures that allows comparisons to be made among various proposed distributed commit algorithms. The context for our work is the Highly Available Systems project at the IBM San Jose Research Laboratory [AAFKM83].

INTRODUCTION

In a distributed data base system, the actions of a transaction (an atomic unit of consistency and recovery) may occur at more than one site. The purpose of a distributed commit algorithm is to insure the atomicity of distributed transactions, thereby insuring the consistency of the data bases involved. Consequently, a distributed commit algorithm must guarantee that either all or none of the actions of a transaction take effect, in spite of site and communication link failures. This guarantee can never be absolute (v. [Gray78]). Various proposed algorithms [Gray78, HaSh80, Lamp80, LSGGL80, MoLi83, Skee81, Skee82] offer the guarantee with varying degrees of reliability and varying efficiency.

If we consider a distributed system model consisting of processors and communication links, we can give a gross measure of the reliability of a given algorithm by stating how many concurrent component failures of various types the algorithm can tolerate. We classify failures as failures of **omission** or **commission** depending on whether some action required by the algorithm was not taken or some action not specified by the algorithm was taken. Note that we count the number of component failures (processors or links), not the number of instances of failure. We say a commit algorithm **tolerates** a failure if atomicity of transactions is preserved except, possibly, on failed processors and after detection of failure a failed processor can be brought to a consistent state by a recovery algorithm. These notions will be made more precise in the next section. A typical distributed commit algorithm tolerates any number of failures of omission but cannot tolerate some single failures of commission (e.g. telling some sites to commit the transaction and telling others to abort).

Efficiency of a distributed commit algorithm may be measured by the number of messages sent, the time for completion of the commit processing, the level of parallelism permitted during the commit processing, the number of state transitions required, the time required for recovery from failures, the number of log records written, and the number of those log records that are written synchronously to stable storage. Here, we distinguish between records that must be **forced** to stable storage before processing can continue and records that are allowed to migrate asynchronously to stable storage. In general, these numbers are expressed as a function of the number of sites or processes involved in the execution of the distributed transaction.

With respect to failure resiliency, an important characteristic of commit algorithms is their blocking property. A commit algorithm is said to be **blocking** if the algorithm requires that a process that had consented to committing a transaction wait, after noticing the failure of its coordinator, until it can reestablish communication with its coordinator to determine the final outcome (**commit** or **abort**) of the commit processing for that transaction. Depending on how fast the coordinator site recovers, this wait period could be quite long. During this period the waiting process may hold locks on resources and prevent the completion of otherwise independent transactions. Skeen has observed that no non-blocking distributed commit algorithm can tolerate a number of failures of omission sufficient to partition the network [Skee81].

In this paper, we show how to trade increased message traffic, and (possibly) some delay in committing transactions, for improved failure recovery time, reductions in numbers of synchronous writes to stable storage, reductions in numbers of required acknowledgements, and removal of the blocking property (assuming that network partitioning is impossible). To achieve this, we use new efficient methods for guaranteed multicast of a message or **Byzantine Agreement**. The goal of Byzantine Agreement algorithms is for a set of processors to agree on the contents of a message sent by one of them. Several different Byzantine Agreement algorithms have been developed to take advantage of trade-offs that exist between requirements on number of participants, number of phases, and number of messages [DFFLS82, DoSt82a, DoSt82c, LyFF82]. The name Byzantine is applied because no assumption is made about the behavior of faulty components except to quantify the maximum number of independent failures handled. Failures of omission or commission are tolerated up to the maximum number.

We describe a non-blocking algorithm for distributed commit that tolerates any number of failures of omission up to the number required to partition the network, and tolerates a similar number of failures of commission that occur sufficiently late in commit processing. We emphasize that we are not supplying Byzantine reliability for transaction processing as a whole, but are only providing Byzantine reliability during commit processing. However, we will handle the case in which some processes are told to commit while others are told to abort. Moreover our methods suggest ways in which greater Byzantine reliability (greater tolerance for even arbitrary failures of commission) may be provided for transaction processing as a whole.

HIGHLY AVAILABLE SYSTEMS ENVIRONMENT

In this section we describe the context for our work.

A loosely connected set of processors intended to provide high availability for data base subsystems is called a **cluster**. The connectivity and communication bandwidth within a cluster are assumed to be high. In the Tandem NonStop™ architecture such a cluster is called a **node** [Borr80]. We assume that our network is a connected collection of clusters. Some clusters may contain single processors but the typical cluster contains three or more processors. The connectivity and communication bandwidth among different clusters are assumed to be low.

For each distributed transaction, a tree of processes is created to do its work. This multi-level (not just two-level) tree-of-processes model of distributed transaction execution is the model adopted by the distributed data base systems R* [DSHLM82, HSBDL82, LHMWY83] and ENCOMPASS [Borr81]. Each process may access a different data base. Under normal circumstances a process executes on only one processor. More than one process in the tree may be located on the same processor. Initially, only the root of this tree, which is called the **transaction coordinator** process, exists. It is this process which is connected to the user at a terminal or an application program executing a transaction. The tree evolves as the transaction execution proceeds. The edges represent the superior/subordinate relationship between processes. Non-root, non-leaf nodes act as coordinators as well as subordinates, while leaf nodes act only as subordinates and the root node acts only as coordinator. Each process knows about and communicates directly with only its immediate neighbors (i.e., immediate descendents and ancestor, if any) in the tree.

The cluster that contains the root of the tree of processes is called the **root cluster** of the transaction. All the other clusters spanned by the tree are called **non-root** clusters. Figure 1 shows an example process tree, in which C is the root, and C, B, F and G belong to the root cluster.

In the Highly Available Systems (HAS) prototype at IBM San Jose Research Laboratory [AAFKM83], which provides the context for our work, each processor will run one or more data base (DB) subsystems, each managing a data base, as well a communication subsystem. A DB subsystem is completely responsible for managing its data base. Mirrored copies of the data base files are maintained. In case a DB subsystem fails, a backup subsystem **in the same cluster** is activated to take over the responsibilities of the failed system. The backup DB subsystem will also have access to the primary DB subsystem's files (including logs).

We assume that each data base subsystem has a **log** in stable storage, which is used to recoverably record the data base or transaction state changes. We also assume the existence of a facility, independent of transaction processing, for reaching Byzantine Agreement. This facility includes, on each processor, an **agreement log**, which contains a history of recent agreements (accessible by their times of generation). Old agreements are assumed archived, and recent agreements are assumed to be held in virtual storage. The Byzantine Agreement algorithm, as used by our commit protocols, is performed by the communication subsystems of all the processors within a cluster; it is **not** performed between clusters. The agreement is not carried out across clusters for two reasons: (1) the inter-cluster

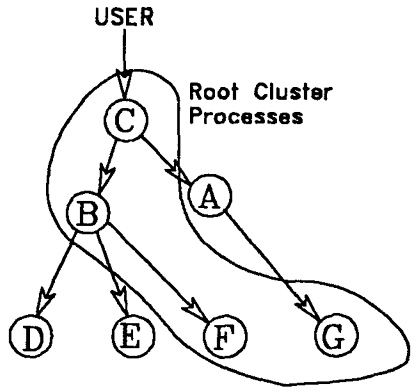

Figure 1: Example Process Tree with Root Cluster Processes Identified

connectivity is not likely to be high enough to provide good Byzantine reliability, and (2) the cost of system-wide agreement may be prohibitively high. When an agreement is reached, appropriate messages are delivered by each processor's communication subsystem to a list of **mailboxes** (local to the processor) that were included in the agreement message. The Byzantine reliability (number of independent faults tolerated by the modified commit algorithm) will be min(conn-1,card), where **conn** is the connectivity of the root cluster, and **card** is the cardinality of the root cluster.

DISTRIBUTED TRANSACTION COMMIT

In this section we describe one of the R* commit protocols which we modify with Byzantine Agreement in the following section.

The distributed commit algorithms referenced in the introduction are variations of what has come to be known as the two-phase commit algorithm [Gray78, Lamp80]. In the latter, during the first (or prepare) phase, all the participating processes of the transaction are polled to determine whether they are willing to commit the transaction, by the sending of *prepare* messages, which are answered with *yes/no vote* messages. The poll is initiated by the transaction coordinator process. During the second (or commit) phase, a decision based on the votes received is propagated to all the relevant participants, by the sending of *commit/abort* messages, which may be acknowledged by the sending of *ack* messages.

The traditional two-phase commit algorithm has been extended to define two types of commit algorithms for R*. They are called **Presumed Abort (PA)** and **Presumed Commit (PC)** [MoLi83]. The two protocols differ in their message and log overhead costs. Both the algorithms permit individual processes of a transaction to unilaterally decide to abort the transaction, provided that the process making the abort decision has not already expressed its willingness to commit the transaction.

If the user decides to abort (respectively commit) the transaction, then the root process receives the ***ABORT TRANSACTION*** (respectively ***COMMIT TRANSACTION***) command. The root propagates the ***abort*** to its subordinates. They in turn propagate it to their subordinates, and so on. ***Abort*** messages sent under these circumstances are not acknowledged.

If the root receives a ***COMMIT TRANSACTION*** command, it initiates the commit algorithm by sending *prepare* messages to its subordinates. At that time the root can choose which algorithm it wants to follow: PA or PC. In both the algorithms, after receiving a *prepare*, a process forwards the *prepare* to its subordinates, if any. If PC is chosen, a **collecting** record, which contains the names of the immediate subordinates, is forced to stable storage before the prepare messages are sent. For brevity, we will discuss and modify only PA. Application of our techniques to PC is straightforward. What follows is a pseudocode description of PA. Local procedure calls, which are synchronous, are distinguished from remote requests to perform procedures. Remote requests are made by sending messages to the remote processes, and the performance of the requested actions is asynchronous with respect to the requestor.

```
/* User has requested COMMIT TRANSACTION for transaction T, for
   which C is transaction coordinator */
PA: PROCEDURE(T) AT SITE(C);
BEGIN
  /* Phase 1 */
  PERFORM PrepareTrans(T) AT SITE(C);  /* same site */
  WAIT UNTIL "reply is received";
  IF "reply is <Vote yes on T>"
    THEN /* Phase 2 */ CALL CommitTrans(T); /* AT SITE(C) */
```

```
END PA;


/* For phase 1 of 2-Phase Commit */
PrepareTrans:   PROCEDURE(T) AT SITE(S);
BEGIN
  IF "S is not willing to Commit T"
    THEN okay := FALSE
    ELSE BEGIN  /* S is willing to Commit T */
      okay := TRUE;
      FORALL S1 IN Subordinates(S) DO
        SEND(<PERFORM PrepareTrans(T) >) TO S1;
      WHILE okay AND "some subordinate has not replied" DO
        BEGIN
          WAIT for a reply from some subordinate;
          IF "reply is <Vote no on T>"
            THEN okay := FALSE;
        END;  /* while */
    END;  /* S is willing to Commit T */
  IF okay
    THEN BEGIN   /*all subordinates willing to commit */
      Log(<Prepared T>) and force record to stable storage;
      REPLY(<Vote yes on T>);
    END
    ELSE BEGIN
      REPLY(<Vote no on T>);
      CALL AbortTrans(T);
    END;
END PrepareTrans ;


/* For phase 2 of 2-Phase Commit */
CommitTrans:   PROCEDURE(T) AT SITE(S);
BEGIN
  Log(<Committed T>) and force record to stable storage;
  IF "S is the transaction coordinator, C"
    THEN Tell User that T was Committed
    ELSE REPLY(<Acknowledge Commit of T>);
  FORALL S1 IN Subordinates(S) DO
    SEND(<PERFORM CommitTrans(T) >) TO S1;
  COMMIT T locally;
  WHILE "some subordinate has not replied" DO
    WAIT for a Commit Acknowledgement from some subordinate;
  Log(<Ended T>);
END CommitTrans ;


AbortTrans:   PROCEDURE(T) AT SITE(S);
BEGIN
  Log(<Abort T>);
  FORALL S1 IN Subordinates(S) DO
    IF "S1 didn't Vote no on T"
      THEN Send(<PERFORM AbortTrans(T) >) TO S1;
    ABORT T locally;
    IF "S is the coordinator, C"
      THEN Tell User that T was Aborted;
  END AbortTrans ;
```

More detailed descriptions of the PA and PC protocols, the recovery algorithms and performance comparisons can be found in [MoLi83]. The phrase "COMMIT T locally" ("ABORT T locally") means "make all updates visible and free locked resources" ("dispense with all updates and free locked resources"). The logging of COMMIT or ABORT by the root process must guarantee the associated action (both locally and by the subordinates), whether or not there are failures.

When a subordinate DB subsystem recovers from a failure, it may have some in doubt transactions (those for which prepare but no commit or abort records are logged). For each in doubt transaction the subordinate must ask its superior (its immediate predecessor in the tree) for the resolution (commit or abort). If the superior finds no record of the transaction then it presumes abort in PA.

MODIFIED COMMIT ALGORITHM

In this section, we describe the distributed commit algorithm using Byzantine Agreement (BPA) as a modification of PA. It is also possible to define the (similar) BPC protocol by modifying the presumed commit protocol PC.

We assume that before it initiates BPA the transaction coordinator (C) has collected information about the depth of the tree of processes generated, and consequently has a good upper bound estimate (*Phase1Time*) on the time required to complete the prepare phase. (If this bound is exceeded, then C must abort the transaction, and may retry it using a larger bound on the prepare phase.) What follows is pseudocode description of BPA.

```
  /* User has requested COMMIT TRANSACTION for transaction T, for which C is
     transaction coordinator. C already has determined Phase1Time. ByzT
     is a bound on the time for performing Byzantine Agreement, which should include an
     allowance for clock differences and network problems [DoSt82b, LaMe82]. */

  PA: PROCEDURE(T) AT SITE(C);
  BEGIN
    StartPhase1 := ClockTime; /* Time from local clock */
    ExpectEndPhase1 := StartPhase1 + Phase1Time;
    ExpectEndPhase2 := ExpectEndPhase1 + ByzT;
    /* Phase 1 */
    PERFORM PrepareTrans(T,ExpectEndPhase2) AT SITE(C); /* same site */
    WAIT UNTIL "reply is received";
    EndPhase1 := ClockTime;
    IF EndPhase1 <=ExpectEndPhase1 AND
              "reply is <Vote yes on T> WITH Mailboxes"
      THEN BEGIN
        APPEND Mailbox(C) TO Mailboxes;
        Log(<Prepared T>) and force record to stable storage;
        /* Asynchronously attempt Byzantine Agreement for processors in Cluster(C) */
        ATTEMPT ByzantineAgree(T,Mailboxes);
        /* Action will be taken at all cluster processors based on whether
           or not agreement is reached before the timer goes off */
      SetTimer(Mailbox(C),ExpectEndPhase2)    AND  WAIT;
      /*  Wait until waked by timer popping or by commit in Mailbox   */
      IF "waked by timer"
        THEN CALL AbortTrans(T)   /*  at Site (C)   */
        ELSE BEGIN  /*  waked by commit message in Mailbox(C)   */
          UnSetTimer(Mailbox(C));
          CALL CommitTrans(T);   /*  at Site (C)   */
        END;  /*  waked by commit   */
    END  /*  EndPhase1 <=ExpectPhase1  AND reply is yes   */
    ELSE CALL AbortTrans(T);   /*  at Site (C)   */
END PA;


/*  For phase 1 of 2-Phase Commit   */
PrepareTrans:   PROCEDURE(T,ExpectEndPhase2)   AT SITE(S);
BEGIN
  IF "S is not willing to Commit T"
    THEN okay := FALSE
    ELSE BEGIN   /*  S is willing to Commit T   */
      okay := TRUE;
      Mailboxes := EmptyList;
      FORALL S1 IN Subordinates(S)  DO
        SEND(<PERFORM PrepareTrans(T,ExpectEndPhase2) >) TO S1;
      WHILE okay AND "some subordinate has not replied" DO
        BEGIN
          WAIT for a reply from some subordinate;
          /*  Replies have the form: vote WITH NewMailboxes   */
          IF "vote is <Vote no on T>"
            THEN okay := FALSE
            ELSE APPEND NewMailboxes TO Mailboxes;
        END;  /*  while   */
    END;  /*  S is willing to Commit T   */
  IF okay
    THEN BEGIN
      Log(<Prepared T>) and force record to stable storage;
      IF "S is in Cluster(C) "
        THEN APPEND Mailbox(S) TO Mailboxes;
      REPLY(<Vote yes on T> WITH Mailboxes);
      IF "S is not the coordinator, C" AND "S is in Cluster(C) "
        THEN BEGIN
          SetTimer(Mailbox(S),ExpectEndPhase2)   AND  WAIT;
          /* Wait until waked by timer popping or by commit in Mailbox   */
          IF "waked by timer"
            THEN CALL AbortTrans(T)   /*  at Site (S)   */
            ELSE BEGIN  /*  waked by commit message in Mailbox(S)   */
              UnSetTimer(Mailbox(S));
              CALL CommitTrans(T);   /*  at Site (S)   */
            END;  /*  waked by commit   */
        END;  /*  S is in root cluster and is not the coordinator   */
    END  /*  if okay   */
    ELSE BEGIN
      REPLY(<Vote no on T> WITH EmptyList);
      CALL AbortTrans(T);
    END;
END PrepareTrans ;


/* On each PROCESSOR in Cluster(C), if the Byzantine Agreement succeeds, the
   Communications Subsystem on that Processor should perform the following procedure */
ByzSucceed:  PROCEDURE(T,Mailboxes)  AT PROCESSOR(P);
BEGIN
  LOG(<Committed T>) IN BYZANTINE LOG;
  FORALL S RUNNING ON P SUCH THAT Mailbox(S) IS IN Mailboxes DO
    POST(<Commit T>) TO Mailbox(S);  /* wakes S */
END ByzSucceed ;


/* For phase 2 of 2-Phase Commit */
CommitTrans:  PROCEDURE(T)  AT SITE(S);
BEGIN
  IF "S is the coordinator, C" OR "S is not in Cluster(C) "
    THEN DO
      Log(<Committed T>) and force record to stable storage;
      IF "S is the coordinator, C"
        THEN Tell User that T was Committed
        ELSE REPLY(<Acknowledge Commit of T>);
    END;
  IF "S is in Cluster(C) " AND "S has some subordinate NOT IN Cluster(C) "
    THEN Log(<Committed T>) and force record to stable storage;
  FORALL S1 IN Subordinates(S) AND NOT IN Cluster(C) DO
    SEND(<PERFORM CommitTrans(T) >) TO S1;
  COMMIT T locally;
  WHILE "some subordinate NOT IN Cluster(C) has not replied" DO
    WAIT for a Commit Acknowledgement from such a subordinate;
  Log(<Ended T>);
END CommitTrans ;


AbortTrans:  PROCEDURE(T)  AT SITE(S);
BEGIN
  Log(<Abort T>);
  FORALL S1 IN Subordinates(S) AND NOT IN Cluster(C) DO
    IF "S1 didn't Vote no on T"
      THEN Send(<PERFORM AbortTrans(T) >) TO S1;
  ABORT T locally;
  IF "S is the coordinator, C"
    THEN Tell User that T was Aborted;
END AbortTrans ;
```

Phase 1 (PrepareTrans) in BPA differs from PA in two ways. One difference is that every process forwards, with the Perform PrepareTrans message, the time (supplied by C) by which the first phase is to terminate. The other difference involves mailboxes for the processes that are in C's cluster, the root cluster. A process that votes yes should append to its vote message a list of all mailbox ids that it received with the yes votes of its subordinates, and should append a mailbox id of its own if it is in the root cluster. (In the example of Figure 1, assuming that all processes vote yes, B will forward along with its yes vote the id of its mailbox and that of F's.) Each process that is in the root cluster waits to receive (in the mailbox whose id it appended) an agreement to commit, setting a timer allowing enough time for one Byzantine Agreement after the upper bound on termination of phase 1. If the commit agreement is received before the timer expires, then the process commits immediately. If the timer expires, then the process aborts instead.

During phase 1 of BPA, processes outside the root cluster behave just as they do in PA, except that they must forward mailbox ids. (In the example of Figure 1 A will forward the id of G's mailbox.) Outside the root cluster, BPA is blocking, since processes do not set timers. (In the example of Figure 1 D, E and A do not set timers.) Thus a process that has voted yes may wait indefinitely to be told by its superior whether to commit or abort the transaction. (In a system that has back-ups for database systems, such as the HAS prototype, a process must wait until a back-up for its superior is activated.) However inside the root cluster, processes stay in the prepared state at most until the timeout, so BPA is not blocking inside the root cluster.

At the end of the phase 1 of BPA, the transaction coordinator C decides whether or not to commit the transaction. C decides to commit only if all its subordinates voted yes. If C decides to abort, it aborts the transaction locally, but sends no messages to subordinates in the root cluster; these processes will abort the transaction when their timers expire. If C decides to commit, it behaves like a subordinate in PA, writing a prepared record and forcing it to stable storage. (The prepared record is for the purpose of recovery.) C also appends a mailbox id to the list of root cluster mailbox ids received from subordinates. It then generates a Byzantine Agreement on the commit message (among all the processors in the root cluster, not just those that participated in the transaction) and waits on its mailbox. The mailbox ids are sent as part of that commit message, but are not used until the commit point is reached. The **commit point** is the point at which Byzantine Agreement is reached; C should force a commit record to its log when it is informed by a message in its mailbox that agreement has been reached. When a processor where one of the mailboxes is located reaches the commit point, it posts a commit agreement message to that mailbox (or to those mailboxes, since more than one may be on a single processor).

If a root cluster process S receives the commit agreement before its timer expires, then it commits the transaction locally, performing updates and dropping locks. Only if S has any subordinates that are not in the root cluster must it write (and force) a commit record to the log. In that case, S must also complete processing as in phase 2 of PA for those subordinates.

If a root cluster process times out before receiving the commit agreement, then it acts as if it had been told to abort the transaction. The transaction coordinator C may timeout just like any other process. A process that times out needs to send abort messages only to its subordinates in non-root clusters.

RECOVERY

This section briefly discusses some of the techniques for cluster and processor recovery in a system using BPA.

CLUSTER RECOVERY

Clusters are not supposed to crash. If a cluster crashes we call this event a **catastrophe**. The details of cluster recovery are beyond the scope of this paper. However, we can sketch the required operations by treating a cluster like a single processor: the data bases managed by a cluster must be recovered from checkpoints and the logs of all processors in the cluster must be examined to obtain a consistent state assignment for each transaction. If any processor has logged a transaction as committed, then all must agree to that assignment.

PROCESSOR RECOVERY

When a member processor of a cluster recovers from a crash or from a cluster partition in which it was not in the majority partition, it copies the agreement logs of t processors and takes the union of their successful agreements to form its agreement log. (Here t is the Byzantine reliability.) If t processors are not available, then all available processors in the cluster are used. Only processors that can communicate with a majority of the processors in the cluster are allowed to perform distributed actions. When a processor finds itself in a non-majority partition it suspends all distributed actions until it becomes part of a majority partition, at which time it undergoes Processor Recovery.

PROCESS RECOVERY

When a DB subsystem recovers, it finds the results of agreements pertaining to transactions in the prepared state before failure in the agreement log in virtual storage on the processor on which it now resides. Neither log read operations nor sending of messages to other processors is required to resolve these **in doubt** transactions. (Such operations would be required in the case of PA.) This speeds the recovery procedure for the DB subsystem, and allows it to resume operations quickly. Operations resumed include informing subordinates of the results of commit processing, as required by PA, if the subordinates are outside the cluster.

FAILURE CLASSIFICATION

In order to be able to compare the reliability characteristics of different commit protocols, in this section we introduce a model of failure classification.

Our classification of failures and our notions of degrees of reliability are defined with respect to a fixed decomposition of the distributed system into components. These components are treated as black boxes that either do or do not meet a predetermined set of input/output specifications. For example, we may decompose a distributed system into a set of processors, terminals, disks, and unidirectional communication links. Alternatively, we could view all the processors and all the communications links as one component and specify the actions (terminal and disk reads and writes) that this component is to perform given the behavior of the other components. An event (say the loss of a message between processors) may be an error (failure to meet specifications) with respect to the former decomposition without causing any error with respect to the latter. In this case, we say the event is **tolerated** with respect to the latter decomposition.

More formally, we postulate a model in which each component has a set IE of possible input events, a disjoint set OE of possible output events, and a correctness specification providing a relation between inputs and outputs so that any subset of the total set of events (the union of IE with OE) may be classified as **correct** or **in error**. We emphasize here that we are discussing classification **not detection** of errors.

If some collection of events C is in error but there exists a set of **output events** D such that the union of C with D is correct, then we say that C is an error of **omission**. Otherwise, we say that C is an error of **commission**.

It is straightforward to add time explicitly rather than implicitly and talk about partial orderings instead of sets of events. The important point is that the notions of omission, commission, and toleration can be made precise relative to any decompositions for which the notion of error can be made precise.

For the remainder of the paper, we use the two decompositions described in the first paragraph of this section. We sketch these decompositions informally, but we assert that the notions presented can be made rigorous. The errors with which we are concerned for the first decomposition correspond to failure to meet our algorithmic specifications in the case of processors, and failure to faithfully deliver input messages in order in the case of communication links. However, the notion of error in the second decomposition is not the straightforward one involving consistency of the data bases. Instead, it is based on a combination of this straightforward notion with the notion of error defined for the first decomposition. Consider a particular duration of time such as the time to complete commit processing using one of our algorithms. We say that a terminal or disk is **inviolate** if it was not written to by or via any component in error (with respect to the first decomposition). We say that the processing component is **correct** during this period with respect to the second decomposition if it does not introduce any data base inconsistency to inviolate components. Thus we say that an algorithm **tolerates** an error with respect to the first (finer) decomposition if no error with respect to the second decomposition is introduced because inviolate components remain consistent. Of course, we also wish to require of our algorithms that transactions involving inviolate components make progress toward a successful commit; otherwise, we could achieve absolute reliability by always aborting transactions. This completes our somewhat philosophical sketch of the basis on which we compare the reliability of algorithms.

For definiteness, let the duration with respect to which we define "inviolate" be the duration during which a specific transaction is processed (including any recovery processing on failed processors). We assume that any error is eventually detected and corrected so that database inconsistencies introduced to a terminal or disk by a local error are removed or repaired. We assume that log errors are repaired and the database is brought to a state consistent with the log using PA or BPA recovery algorithms for **in doubt** transactions. With these assumptions we can strengthen our notion of toleration of error: no inconsistency is introduced among inviolate components and eventually all components are brought to consistency by recovery processing if necessary.

It should be clear that errors of omission are much less dangerous than errors of commission. In fact standard two phase commit algorithms are designed to handle errors of omission but not commission. One way to restrict the set of errors that must be tolerated to those of omission only is to implement failstop processing [Schn83]. Failstop components can be implemented so that as long as the number of errors of any kind within each component stays below a fixed bound (cf. Byzantine reliability), the components stop after any internal error. Here the errors of omission possible are of a very restricted variety. It may be possible to implement components that never have errors of commission but allow more general errors of omission and are not so expensively redundant as failstop components.

SAFETY COMPARISON

In this section we make the very strong assumption that the only way for a distributed database to become inconsistent is for some component to commit a transaction that is aborted by another. Thus we ignore any error that might, for example, have a transaction seen by the wrong sites. We are interested only in the safety of distributed commit processing, and assume the absolute safety of all other aspects of distributed transaction processing.

Theorem 1. The set PA of commit and recovery algorithms tolerates any number of failures of omission.

Outline of Proof: Suppose otherwise, that as a result of some number of failures of omission and no errors of commission, a database inconsistency is introduced to inviolate or recovered components. Without loss of generality, we may assume that the results of a single transaction T are recorded as committed at one component and as aborted at another. We may further assume that these components are written to by two separate processes p and q: say p correctly aborts and q correctly commits T. In PA process p may abort T only if (1) p has voted No on T, (2) p received "Perform AbortTrans(T)" from its immediate predecessor, (3) p is the coordinator for T and it sends "Perform AbortTrans(T)" to each of its subordinates that did not vote No on T, or (4) p recovered with T **in doubt** and its immediate predessor had no record of T. An analysis of each case under the assumption of no errors of commission shows that the coordinator for T cannot have sent "Perform CommitTrans(T)", so when q committed T it was an error of commission, contradicting our assumption. □.

Theorem 2. Within a cluster, the set BPA of commit and recovery algorithms tolerates any number of errors of any kind up to the Byzantine reliability of the Byzantine Agreement algorithms it calls.

Outline of Proof: As above we assume otherwise, that process p correctly aborts and process q correctly commits T. In BPA, q may commit T only if (1) q receives a successful Byzantine Agreement to commit T or (2) q recovers with T **in doubt** and finds a successful Byzantine Agreement to commit T in the agreement log. It is a property of the Byzantine Agreement algorithms that if a successful Byzantine Agreement appears in any processor's log, then it appears in every correct processor's log, provided no more than t errors have occurred (where t is the Byzantine reliability). Thus if p is correct or correctly recovers, p must have a successful Byzantine Agreement to commit T in its agreement log. Moreover, this agreement must have been received before the time that p's timer would have wakened it if p had been operating correctly. Thus p cannot have correctly aborted T, contradicting our assumption. □.

DISCUSSION

When the root cluster is viewed as a single highly available processor, our modified commit algorithm BPA has the same reliability as the algorithm we modify: it tolerates any number of failures of omission and cannot tolerate certain single errors of commission. In fact the message traffic outside the root cluster is unaltered, as are most of the other measures of efficiency previously discussed. From this external point of view our improvement is in recovery time for systems within the root cluster, and consequently in the length of time resources must be held locked because PA is blocking.

Transactions entirely within the root cluster have the advantage that BPA is non-blocking, in addition to the recovery time advantage. Moreover, BPA tolerates as many errors of commission during its commit phase (i.e., the second phase) as the Byzantine Agreement algorithm on which it depends, since the commit phase consists of a specific period of waiting for a Byzantine Agreement to commit. However, BPA cannot tolerate omission failures sufficient to partition the cluster.

To compare the advantages and disadvantages of BPA and PA, we consider a committing transaction spanning a single cluster. First, let us examine the no failure scenario. BPA saves us the writing of the commit .record (which is forced in the case of PA) and sending of commit acknowledgement messages by all the non-root processes. Furthermore, no commit messages are sent by the non-leaf processes. Against these advantages, BPA has the following disadvantages. The messages sent during the process of attaining the Byzantine Agreement do not have any counterparts in PA. These messages contribute not only network traffic, but also work load (handling of messages and writing the Byzantine Agreement log record) at all the processors in the cluster. Depending on its position in the transaction tree, a given process may get to know about the decision to commit after a longer time in BPA than in PA, so locks may be held longer in BPA. BPA requires the (force) writing of a prepared record by the transaction coordinator process. No such record is written by that process in PA.

Now let us consider the failure scenario. If a superior process fails after one of its subordinates had voted yes, then in PA (running with back-ups, as in the HAS prototype) the subordinate has to wait until the superior is brought up in a backup processor to find out the commit decision. Compared to this, in BPA the failure of any processor in the cluster could increase the time it takes for a process to find out the commit decision. It is quite difficult to compare these delays. On the other hand, if a particular process crashes while in the prepared state, then when that process is brought up in the backup processor with PA, it has to communicate with its superior process to find out the commit decision. But with BPA, it need only examine the local Byzantine Agreement log to find out the commit decision. In this case, BPA is better than PA. But, BPA also imposes the requirement that whenever a processor is brought up after a crash, time must be spent in getting an up-to-date version of the Byzantine Agreement log before the processor can start hosting data base subsystems.

When PA is used, and we consider only the commit protocol operation nothing special needs to be done if network partitioning occurs. On the other hand, when using BPA, network partitioning must be detected and dealt with as described in the section on recovery.

It is beyond the scope of this paper to provide Byzantine reliability to the entire environment of distributed transaction processing. However, we can suggest modifications, at the cost of additional waiting time between receipt of the Byzantine Agreement to commit and the actual commit point, that will tolerate failures of commission during either phase of commit processing. Assuming the existence of an authentication protocol as used in efficient algorithms for Byzantine Agreement [DoSt82c], we require that all messages during the prepare phase be signed with unforgeable signatures and stamped with an unalterable time stamp. The coordinator must include in the commit message all the signed votes of the participants. If any participant detects a discrepancy between the prepare phase message it received and the copy included in the agreement to commit, it transmits this discrepancy to all participants via a second Byzantine Agreement. Finally, each participant waits for a time sufficient for two Byzantine Agreements after the time stamped on the Byzantine Agreement to commit. If no refutation has been received by this time, it then commits the transaction.

BPA assumes that abort messages received in the root cluster are ignored. We can improve the time to abort by allowing abort messages (signed as above) and using them in Byzantine Agreements of refutation as above.

Note that both modifications only require extra messages in case of a failure of commission, but they do require extra waiting time before the transaction can be committed even when there are no failures. Thus they still represent performance improvements over the algorithm presented in [DoSt82b] where Byzantine Agreement is used for both phases of commit processing.

In this paper, we have provided a new kind of trade-off in reliability and efficiency of distributed commit algorithms. Our algorithm is particularly suited to the environment of highly available clusters of processors. It provides a speed of data base system recovery that facilitates cutover from an ailing primary system to a backup system within the cluster, maintaining in many cases the image of a system that has had no failure, even in spite of multiple concurrent failures of omission and commission. While PA and PC have been implemented in R*, our modified protocols have not been implemented.

REFERENCES

AAFKM83 Aghili, H., Astrahan, M., Finkelstein, S., Kim, W., McPherson, J., Schkolnick, M., Strong, R. "A Highly Available Database System", IBM Research Report RJ3755, January 1983.

Borr81 Borr, A. "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing", *Proc. International Conference on Very Large Data Bases*, September 1981.

DFFLS82 Dolev, D., Fischer, M., Fowler, R., Lynch, N., Strong, H.R. "An Efficient Byzantine Agreement Without Authentication", *Information and Control*, to appear. See also IBM Research Report RJ3428, March 1982.

DoSt82a Dolev, D., Strong, H.R. "Polynomial algorithms for multiple processor agreement", *Proc. 14th ACM SIGACT Symposium on Theory of Computing*, May 1982. See also IBM Research Report RJ3342, December 1981.

DoSt82b Dolev, D., Strong, H.R. "Distributed Commit with Bounded Waiting", *Proc. Second IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, July 1982. See also IBM Research Report RJ3417, 1982.

DoSt82c Dolev, D., Strong, H.R. "Authenticated Algorithms for Byzantine Agreement", *SIAM Journal on Computing*, to appear. See also IBM Research Report RJ3416, March 1982.

DSHLM82 Daniels, D., Selinger, P., Haas, L., Lindsay, B., Mohan, C., Walker, A., Wilms, P. "An Introduction to Distributed Query Compilation in R*", *Proc. Second International Symposium on Distributed Data Bases*, Berlin, September 1982. Also IBM Research Report RJ3497.

Gray78 Gray, J. "Notes on Data Base Operating Systems", in *Operating Systems - An Advanced Course*, Lecture Notes in Computer Science, Volume 60, Springer-Verlag, 1978.

Gray81 Gray, J. "The Transaction Concept: Virtues and Limitations", *Proc. Seventh Int. Conf. on Very Large Data Bases*, October 1981.

HaSh80 Hammer, M., Shipman, D. "Reliability Mechanisms for SDD-1", *ACM Transactions on Data Base Systems*, December 1980.

HSBDL82 Haas, L.M., Selinger, P.G., Bertino, E., Daniels, D., Lindsay, B., Lohman, G., Masunaga, Y., Mohan, C., Ng, P., Wilms, P., Yost, R. "R*: A Research Project on Distributed Relational DBMS", *Database Engineering*, Volume 5, Number 2, December 1982. Also IBM Research Report RJ3653, October 1982.

LaMe82 Lamport, L., Melliar-Smith, P.M. "Synchronizing Clocks in the Presence of Faults", Technical Report, SRI International, March 1982.

Lamp80 Lampson, B. "Atomic Transactions", Chapter 11 in *Distributed Systems - Architecture and Implementation* , B. Lampson (Ed.), Lecture Notes in Computer Science Vol. 100, Springer Veralg, 1980.

LHMWY83 Lindsay, B., Haas, L., Mohan, C., Wilms, P., Yost, R. "Computation and Communication in R*: A Distributed Database Manager", To Appear in *Proc. 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, October 1983. Also IBM Research Report RJ3740, January 1983.

LSGGL80 Lindsay, B., Selinger, P., Galtieri, C., Gray, J., Lorie, R., Putzolu, F., Traiger, I., Wade, B. "Single and Multi-Site Recovery Facilities", In *Distributed Data Bases*, Edited by I.W. Draffan and F. Poole, Cambridge University Press, 1980. Also Available as "Notes on Distributed Databases", IBM Research Report RJ2571, San Jose, July 1979.

LyFF82	Lynch, N., Fischer, M., Fowler, R. "A Simple and Efficient Byzantine Generals Algorithm", *Proc. Second IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, July 1982.
MoLi83	Mohan, C., Lindsay, B. "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", *Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983. Also IBM Research Report RJ3881, June 1983.
PeSL80	Pease, M., Shostak, R., Lamport, L. "Reaching Agreement in the Presence of Faults", *JACM*, Vol. 27, No. 2, pp. 228-234, 1980.
Schn83	Schneider, F.B. "Fail-Stop Processors", *Proc. COMPCON Spring '83*, San Francisco, March 1983, pp. 66-70.
Skee81	Skeen, D. "Nonblocking Commit Protocols", *Proc. ACM/SIGMOD International Conference on Management of Data*, Ann Arbor, Michigan, 1981, pp. 133-142.
Skee82	Skeen, D. "A Quorum-based Commit Protocol", *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1982, pp. 69-90.