論文 / 著書情報
Article / Book Information

Title	An Efficient Commit Protocol Exploiting Primary-Backup Placementin a Distributed Storage System
Author	Xiangyong Ouyang, Tomohiro Yoshihara, Haruo Yokota
Journal/Book name	Proc. of IEEE 12th Pacific Rim International Symposium on Dependable Computing (PRDC2006), Vol. , No. , pp. 238-247
Issue date	2006, 12
DOI	http://dx.doi.org/10.1109/PRDC.2006.17
URL	http://www.ieee.org/index.html
Copyright	(c)2006 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.
Note	このファイルは著者（最終）版です。 This file is author (final) version.

An Efficient Commit Protocol Exploiting Primary-Backup Placement in a Distributed Storage System

Xiangyong Ouyang[†], Tomohiro Yoshihara[†], and Haruo Yokota[‡†]
[†]Department of Computer Science, Graduate School of Information Science and Engineering,
Tokyo Institute of Technology 2–12–1 Ookayama, Meguro-ku, Tokyo 152–8552, Japan
[‡]Global Scientific Information and Computing Center, Tokyo Institute of Technology
2–12–1 Ookayama, Meguro-ku, Tokyo 152–8550, Japan
{ouyang, yoshihara}@de.cs.titech.ac.jp, yokota@cs.titech.ac.jp

Abstract

Advanced data engineering applications require a large-scale storage system that is both scalable and dependable. In such a system, an atomic commit protocol becomes imperative to ensure the consistency and atomicity of transactions. In this paper we present a new commit protocol, BA-1.5PC, which is well tailored to such distributed storage environments as autonomous disks [20] that use a primary-backup storage schema. The protocol achieves an efficient commit process while also guaranteeing a high dependability by combining several approaches: (1) a low-overhead log mechanism that eliminates blocking disk I/Os, (2) removing the voting phase from commit processing to gain a faster commit process, and (3) a primary-backup assisted recovery strategy to enhance dependability in the presence of possible failures, so that a master failure in the decision phase will not block prepared cohorts of a transaction. Experiments were carried out on a trial version of an autonomous disks system to verify its efficiency. The results indicate that this protocol significantly outperforms several well-known commit protocols in terms of transaction throughput.

1 Introduction

Future applications of advanced databases will require scalable, dependable and highly efficient storage systems [12]. Among the many parallel storage systems that have been proposed [12, 15], clusters of share-nothing machines, which share no memory or disk among nodes in the system, show promise for large-scale distributed storage because of their high scalability. *Autonomous disks* [20] (AutoDisk) provides such a solution to scalable distributed storage. An autonomous disks system is configured as a cluster of disk drives interconnected in a network. Data are partitioned into all disks, and these disks receive accesses uniformly via a distributed directory Fat-Btree [21]. Each disk (called a PE hereafter) is equipped with a disk-resident intelligent processor to handle data distribution and load balance. With carefully designed protocols, they are able to tolerate unexpected disk or software failures.

When the distributed storage system expands, the possibility increases that some parts will fail. To improve data availability in the face of disk failures, the Autodisk system adopts chained declustering [9] as a primary-backup storage schema, by which the data on each disk (primary data) are replicated in the logically neighboring PE (backup data), so that when one PE fails, the data stored in that PE is still available as the backup data. Accesses to that part of the data are forwarded to the replica, thus masking the PE failure to user applications. Users can continue to search and update that part of the data despite the absence of a PE in the system, although with possibly degraded performance because of data migration and load balancing for reconfiguration. Acceptable performance degradation is thus achieved despite software/hardware failures at some PEs.

The Fat-Btree directory can provide fast indexing and high retrieval throughput in AutoDisk, but its strategy of partial duplication also adds complexity to structure modification operations (SMOs). With Fat-Btree, index pages are partially copied across PEs. When an insertion of one object into a PE causes a SMO at an index page, all PEs that maintain a copy of that index page must be updated synchronously. As the SMO propagates upward to higher index pages, more PEs become involved in this SMO. Ultimately, an update to the root requires all the PEs in the system to participate in the synchronization. This makes it necessary to borrow the concepts of transactions with their ACID properties, and thus an atomic commit protocol becomes imperative to guarantee the correctness of data in

AutoDisk. For a transaction in AutoDisk (usually an insertion or deletion that can cause a SMO), the PE that initiates the transaction is called the master. All other PEs that own a copy of the affected index pages are cohorts of the transaction.

Unfortunately, existing commit protocols cannot yield satisfactory performance if applied directly to AutoDisk. The best known of all commit protocols is two-phase commit (2PC). However, it can produce severe delays while synchronizing the copies under the primary-backup scheme. 2PC is also prone to blocking cohorts if a master fails in the decision phase. Although there have been many attempts at commit protocols that ameliorate the drawbacks of 2PC, none can fully exploit the features of AutoDisk, nor can they provide the high efficiency and nonblocking properties that are desirable for AutoDisk.

We propose a commit protocol to achieve high efficiency and nonblocking properties by exploiting the characteristics of AutoDisk. This protocol can be applied to any storage system in which primary-backup chained declustering is adopted. This protocol, named backup-assist 1.5-phase commit protocol (BA-1.5PC), borrows from IYV the idea of eliminating the vote phase of commit processing by implicitly including the vote in the acknowledge message from a cohort to the master. Hence, the number of messages exchanged during commit is reduced. Unlike IYV, in which a cohort stores part of its online transaction information (redo log and read locks) to the master, BA-1.5PC eliminates the need for a forced write of the log message by using an improved form of neighbor WAL [10] (nWAL), a natural extension of nWAL in the primary-backup environment, to greatly reduce overhead related to log writes.

BA-1.5PC handles failures effectively by exploiting the primary-backup structure. If a cohort fails, the backup of that cohort can assume its role in the transaction. Log records for operations during the cohort's absence will be maintained by the backup cohort until the failed cohort is fixed and restarts. During its recovery, it can easily restore its state without the participation of the master by reading the complete list of log records at its backup and applying them to bring its data up to date. A highlight of this protocol is that a master failure during transaction processing is also surmounted by its backup. Like cohorts, a master's backup is located in its logical neighbor. If the primary master fails before the decision phase, the backup master will automatically abort the transaction. On the other hand, if the master fails during the decision phase, leaving some cohorts unaware of the final decision, the backup master will assume the master's responsibility, and reinform all cohorts of the result of transaction, instead of forcing other cohorts to wait until the failed master finishes its restart and recovery.

BA-1.5PC significantly improves the performance of commit processing. It also solves the difficult blocking problems caused by a master failure, which is a considerable enhancement to traditional low-overhead commit protocols such as IYV. In addition, BA-1.5PC requires only that participating PEs employ a two-phase locking-based concurrency control protocol [4]. Most existing distributed systems are based on this classical protocol.

The rest of this paper is organized as follows. In Section 2 we briefly review some related research work. We analyze the model and features of the AutoDisk system in Section 3, and then propose the BA-1.5PC protocol in Section 4. Failure recovery strategies are also presented in this section. Experiments that show the performance of this protocol in contrast to some other well-known protocols are described in Section 5. The final section gives our conclusions and some future directions.

2 Related research

Distributed database systems must guarantee the consistency of distributed transactions, i.e., all sites participating in a transaction either unanimously commit or unanimously abort any transaction. Usually this is achieved by applying an atomic commit protocol (ACP) at the end of the transaction. The most widely accepted ACP is two-phase commit (2PC). Although widely used, 2PC is recognized to have two major drawbacks [1]. (1) It imposes a heavy overhead on the transaction commit processing in terms of time delays related to message exchanges and forced log writes, and thus causes substantial delay even in the absence of failures. (2) It can cause serious consequences by putting a cohort in a prepared state after a cohort has voted "Yes" for a transaction in the vote phase [5]. A cohort in the prepared state will be blocked waiting for a decision from the master for an arbitrarily long time if the master fails after issuing the vote requests to cohorts but before broadcasting the decision to cohorts. Moreover, it forces a transaction to abort even after it has been successfully processed if any of its cohorts is unreachable during the voting phase.

There has been much research to overcome the deficiencies of 2PC. Despite the emphases these studies place on different aspects, they can be largely categorized into two classes, although their efforts may partly overlap. One category is protocols that aim to minimize overhead incurred in commit processing, while protocols in the other category attempt to achieve a nonblocking commit protocol.

2.1 Low-cost ACP

The main sources of latency in an ACP can be classified into three types: message exchange overhead, forced log writes, and the convoy effect caused by transactions waiting for resources held by other transactions waiting to commit. The Presumed Abort (PA) protocol and Presumed Commit

(PC) [14] protocol are direct variations of 2PC that try to reduce the number of messages exchanged during commit processing as well as the count of forced log writes by making some assumptions. PA [14] assumes that a transaction is aborted if it is not explicitly committed. With PA, a master requires no acknowledge messages from cohorts for the abort decision message in the decision phase. Neither the master nor the cohorts require a forced write to the decision log for an aborted transaction to reduce the overhead of synchronized disk I/O. PC [14] is a counterpart of PA in which a transaction is assumed to be committed if it has not been explicitly aborted. PC requires acknowledge messages from cohorts to the master only for aborted transactions, and requires a forced write to the decision log only for aborted transactions.

Both PA and PC seek to reduce commit process overhead by reducing acknowledge messages and forced log writes in the decision phase, while the voting phase remains the same as for 2PC. One-phase commit (1PC) goes a step further by removing the explicit voting phase from the commit process. 1PC was first suggested in [6], and several variations have already been proposed. The Early Prepare (EP) Protocol [18] assumes that a cohort is prepared to commit after the master receives an acknowledge message from it for each operation. It implicitly includes the cohort's vote in the acknowledge message to remove the explicit voting phase. Its main drawback is that a cohort must perform a forced write to a log for each operation, which leads to serious disk blocking time. What is more, the master must force a membership log record to a disk when initializing the transaction and each time the membership of that transaction expands.

The Coordinator Log (CL) protocol [19] avoids the cohort's forced log write of each operation by centralizing the cohort's log to the master. However, this violates site autonomy and cannot be practical in real distributed systems. The Implicit Yes Vote (IYV) protocol [2] exploits the performance and reliability properties of gigabit- networked distributed database systems, where the propagation latency will be the dominant component of the overall communication cost while the migration of large amounts of data will not pose a problem. Each cohort combines its redo log and read lock with the acknowledge message of an operation to the master to implicitly mean that the transaction is in a prepared state at the cohort. IYV allows failed cohorts to perform part of the recovery procedure independently of the master, and to resume the execution of transactions that are still active in the system instead of aborting them. Although IYV is well adapted to gigabit- networked DBMS, it is vulnerable in that: (1) it concentrates cohorts' log records to the master, which violates site autonomy and can put a bottleneck in the master; and (2) it increases the risk of blocking, because a master failure during commit processing will block the cohorts until the master's recovery.

There is also Optimistic Commit protocol [7], which focuses on reducing the lock waiting time by lending the locks held by transactions in commit processing. The lending is carefully devised to avoid possible cascading aborts if the lending transactions are aborted. This type of protocol generally has good performance because it reduces the blocking at locks held by prepared transactions.

2.2 Nonblocking ACP

A fundamental problem with ACP is that operational cohorts may become blocked in the event of a master failure that occurs after initializing the voting phase but before making a final decision. During the blocked period, the transactions at operational cohorts may continue to hold system resources such as exclusive locks, preventing other transactions from acquiring these resources to proceed. Those transactions in turn become blocked waiting for some resources to be released. This blocking chain may propagate to cause "cascading blocking".

A number of commit protocols have been designed to attack the fundamental blocking problem. Three-phase commit (3PC) [16, 17] was among the first nonblocking protocols. 3PC introduces a new "buffered phase" between the voting phase and the decision phase. In the buffered phase, a preliminary decision is reached about the result of a transaction. Cohorts can reach a global decision from this preliminary decision even in face of a subsequent master failure. However, 3PC achieves the nonblocking property at the expense of increased communication overhead by an extra round of message exchanges. Moreover, both master and cohorts must perform forced writes of additional log records in the buffered phase.

ACP-UTRB [3] and NB-SPAC [11] implement nonblocking by imposing a strong assumption on the communication model: the reliable uniform broadcast primitive, which claims that if any site, correct or not, delivers a message, then all other correct sites will eventually deliver that message. A master uses the broadcast primitive to disseminate its decision message to all cohorts to guarantee that either all or none of the participants know about the fate of the transaction. Practically, the uniform broadcast can be interpreted as implicitly including a round of message exchanges, and is very expensive in terms of the message delay it introduces. It also compromises the scalability of the protocol because the delay grows with the number of participants.

3 Assumed system model

The choice of commit protocol can impact on a system's performance significantly. Storage systems with primary-

backup declustering, like AutoDisk, impose several requirements on a commit protocol: (1) it can add the least possible overhead to the transaction; (2) it can effectively synchronize the primary-backup; and (3) it can take advantage of the primary-backup hierarchy to handle failures in a transaction. For example, a failure at a cohort should be masked by the backup if possible. In particular, a master failure in the commit phase should not block the prepared cohorts.

In this section we will identify the features of AutoDisk that can potentially benefit the commit protocol. We assume a fail-stop failure model, i.e., a failed PE will cease to function instead of delivering unpredictable abnormal states. Failure is assumed to happen in only one PE at any given time, although we will see later that multiple failures can be dealt with in AutoDisk.

3.1 Chained declustering storage structure

The autonomous disks (AutoDisk) system, proposed in [20], is configured as a cluster of intelligent disks in a network environment. Each disk is equipped with an AutoDisk resident processor that can govern the site autonomy of each disk, so that centralized control within the cluster becomes unnecessary. The AutoDisk system uses active rules to implement such properties as transparent access, skew handling and fault tolerance. Data are partitioned across all disks inside AutoDisk in according with some criteria. AutoDisk employs a primary-backup storage hierarchy similar to chained declustering [9], to improve data availability in the event of failures. Besides the data allocated to each disk (primary data), each disk also maintains a copy of the primary data of its logically previous disk. The backup copy of the data is updated asynchronously to the primary copy to reduce possible synchronization costs. Tasks to maintain consistency between primary and backup copies are performed independently of the hosts.

3.2 Fat-Btree: a highly efficient parallel directory

Fat-Btree was proposed in [21] as a parallel directory structure to ensure high-speed access for distributed database systems like AutoDisk. Fat-Btree can provide an indexing mechanism for fast data retrieval in parallel database systems, and also reduces the synchronization costs between process elements (PE). In Fat-Btree, as illustrated in Figure 1, index nodes closer to the root, which are accessed more frequently, have more copies across PEs in the system, which translates into a faster indexing speed by reducing the frequency of query forwarding. Index nodes closer to the data, which are updated more frequently, have fewer copies to avoid heavy overhead in synchronization while updating multiple copies. Fat-Btree is an ideal indexing directory for AutoDisk in that it achieves a proper balance between fast indexing and low synchronization cost.

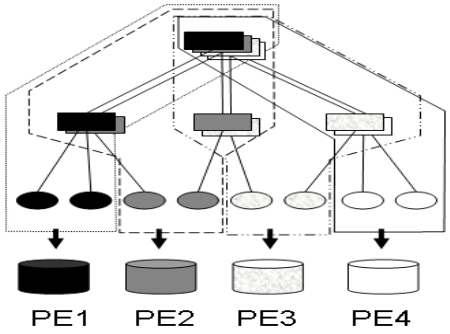

Figure 1. Fat-Btree

3.3 Concurrency control protocol

The correctness and performance of a parallel B-Tree depends heavily on the concurrency control protocol. INC-OPT [13] is a concurrency control strategy for parallel B-Trees. It makes some optimizations of the strict 2PL [4] to achieve a perfect performance in both throughput and response time in the Fat-Btree index structure. Our atomic commit protocol is based on this concurrency control mechanism. MARK-OPT [22], which aims to integrate into INC-OPT some refinement with regard to performance to further improve the concurrency control efficiency, is also eligible to be introduced into our proposal.

4 Low-overhead commit protocol

In this section we will propose the BA-1.5PC protocol. For ease of exposition, we use some specific terms throughout this paper.

Primary data: the whole data range is partitioned across all PEs in AutoDisk. The part of the data allocated to one PE is called primary data for that PE. *Backup data*: each primary data element has a replica in its logically neighboring PE. That copy is named the backup data.

A *master* is the PE where a transaction is initiated. With the Fat-Btree indexing structure, a *cohort* is a PE that has a copy of the index pages that are to be updated in the transaction. Such a cohort is named the *primary cohort*. Besides the operation to update its primary data, a primary cohort must also update its backup data. In such a situation, the PE that stores the backup data is called a *backup cohort*.

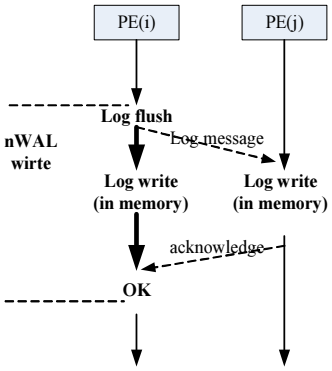

Figure 2. Neighbor WAL

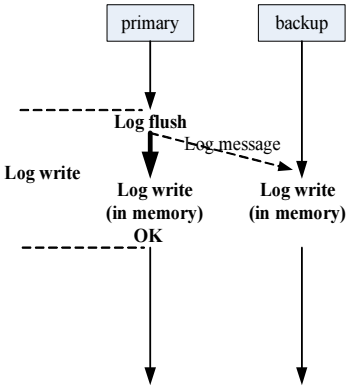

Figure 3. Async-nWAL

4.1 Low-overhead log strategy

Write-ahead log (WAL) is widely accepted as a means for providing atomicity and durability in database systems [8]. WAL asserts that the log records representing changes to data must be stored to stable storage before the new version of data replaces the previous data in stable storage. It is not surprising that a large part of the overhead related to transaction processing arises from forced log writes using WAL, where the transaction is suspended waiting for the blocking write of log records into stable storage to finish. Having recognized this inefficiency, neighbor WAL (nWAL) was proposed to reduce the overhead of logging in parallel databases [10]. Unlike ordinary WAL, nWAL stores the log records in main memory at the local host and several other hosts. As indicated in Figure 2, at a forced log write, PE(i) stores the log records in its main memory and meanwhile transmits the log in a message to its neighbor PE. Thus the same log record has two copies in the neighbors. If PE(i) fails, it can recover to a consistent state with the log stored in its neighbor PE(j). The overhead incurred in nWAL includes the message exchange delay between PE(i) and PE(j) and the time to store log records in main memory. In current high-speed network systems the speed of message delivery is several orders faster than disk I/O, and memory I/O of log records is even faster. nWAL therefore yields lower delays for log writes.

The primary-backup hierarchy in AutoDisk makes it natural to adopt the basic idea of nWAL to reduce log overhead. The original nWAL requires a host to wait for the acknowledgment from its neighbor PE for a log write, which can increase the message overhead during transaction processing. In our protocol, a PE does not wait for the ACK from its neighbor at a log flush operation. In other words, a PE sends the log record to its neighbor asynchronously. The synchronization between a PE and its neighbor is deferred until the final decision phase of a transaction, with the optimistic view that the log message pass will succeed. In the decision phase that terminates a transaction, synchronization between primary and backup will force the backup to complete all its operations if there are any left pending. The commit protocol presented in this paper adopts this asynchronous version of nWAL (async-nWAL), as depicted in Figure 3, to reduce the overhead of logs in transaction and commit processing. With async-nWAL, writing a log will not impose a problem on performance.

It is possible for log records to overflow the buffer no matter how large the memory is. This problem is mitigated in two ways. (1) Log records belonging to completed transactions are deleted from the buffer after commit/abort to reclaim memory. (2) When the log does overflow, some portion of the existing log records are moved to disk files. This can cause a severe delay during a transaction. With modern large volume memories, however, this case is not expected to happen often and therefore presents no problem in practice.

4.2 BA-1.5PC: a low-cost commit protocol

Part of the overhead imposed by a commit protocol comes from the message exchanges during commit processing. We are thus prompted to seek a way to commit a transaction with the least communication overhead possible. The BA-1.5PC (Backup-Assist 1.5 Phase Commit protocol) is such a protocol. As the name indicates, it removes the voting phase from the commit process, and failure recovery is facilitated by the primary-backup structure (replica assist). Figure 4 illustrates the processing flow of this protocol.

When a transaction is initiated in a PE (which becomes the master for that transaction), the master writes a membership log using async-nWAL to its backup PE to mark the start of a transaction. It then begins to deliver operations to cohorts and waits for their replies. On receiving an

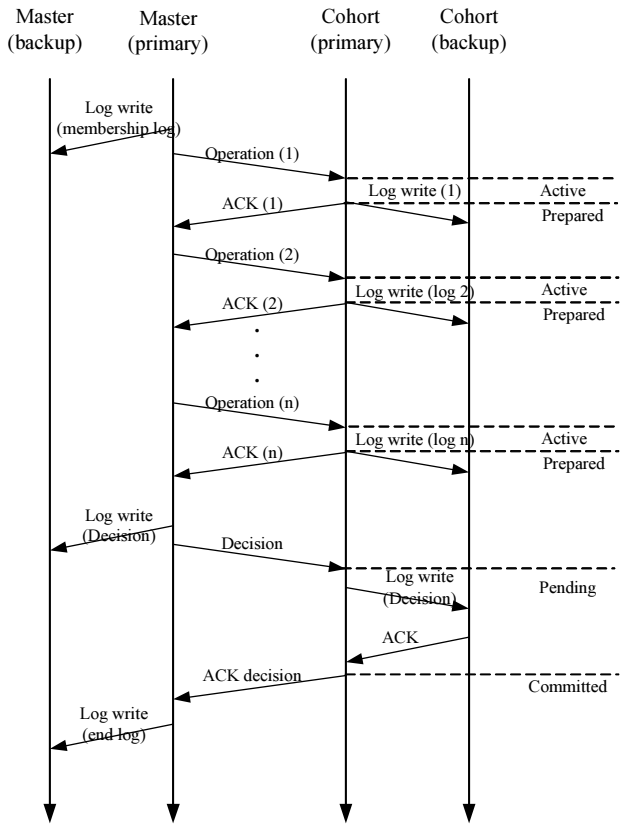

Figure 4. BA-1.5 phase commit

operation command from the master, a cohort executes this operation and issues a log record using async-nWAL to its backup PE including the information about that operation. When the primary cohort finishes the operation, it sends an ACK to the master. Notice that the primary cohort does not wait for the backup cohort to finish its operation to synchronize their data; the optimistic view is that the backup cohort will finish the operation concurrently with the primary cohort. The synchronization is postponed to the end of the transaction. After acknowledging an operation, the primary cohort enters the prepared state from where it can start committing the transaction.

If a cohort decides to abort the operation on its site to resolve either local deadlocks or a local serialization violation, it issues a negative acknowledge (NACK) to the master and unilaterally aborts the transaction locally, writing a log using async-nWAL to its backup cohort. The backup cohort will also abort the effects of that transaction. At every round of operation command/acknowledgment message exchange, if the master detects even one NACK from a cohort, it will abort the transaction and deliver an abort decision to each cohort that has acknowledged with ACK in that round.

When the master has finished all operations of a transaction and received acknowledgments from all cohorts, it makes a final decision about the result of that transaction and forces the decision into the log with async-nWAL. Once past this point, the fate of the transaction has been irreversibly decided, and no single failure can prevent the commitment of this transaction. After that, the master broadcasts the decision to all cohorts and waits for their ACKs. A primary cohort that receives a decision enters the pending state. In the pending state, a primary cohort first writes a decision log with async-nWAL to the backup cohort, committing or aborting the transaction according to the decision, and then suspends while waiting for an ACK from its backup cohort. The pending state is the last opportunity for the backup cohort to bring its data up to date with the primary data. In this period, the backup cohort executes all operations represented by the log records it has received from the primary cohort and accumulated in main memory. Once all the operations are finished, the backup cohort returns an ACK to the primary cohort, which in turn delivers an ACK to the master, signaling the completion of the transaction branch in that cohort. At the transmission of this decision acknowledgment message, both the primary and the backup cohorts can safely delete all log records about that transaction from their main memory to reclaim buffer space. If the primary cohort times out waiting for an ACK from the backup cohort, it concludes that the backup cohort has failed. Instead of waiting until it recovers, the primary cohort directly delivers an ACK to the master without clearing the log records in its memory. These log records are used in the coming recovery process to restore the consistency of the backup cohort. When the master has received ACKs from all cohorts, it writes an end log using async-nWAL to terminate the transaction completely.

The primary cohort experiences a pending phase during the commit process, but the result of the pending state does not actually affect the final outcome of a transaction. This is the origin of the name 1.5-phase commit: a pending phase with a result that can be discarded is counted as 0.5 phase in the commit process. The decision phase counts for the complete phase. The pending state at the primary cohort is necessary because it is the only point at which the primary and backup are synchronized during a transaction. Our low-overhead log strategy has already excluded the synchronization from message deliveries from primary to backup in pursuit of higher performance, with the optimistic view that the operations on the backup will succeed. Without a pending phase in the decision phase, a failure in the backup can never be detected during a transaction, and the primary will continue its normal processing and delete the log records of the completed transaction from its memory, making it impossible for the backup to restore its data after it recovers from a failure.

A more aggressive way exists to eliminate the 0.5 phase

from the decision phase. The primary cohort may send its acknowledgment directly to the master upon receiving a decision, without synchronizing with the backup cohort. Doing so can further reduce the delay in commit processing. However, this method trades less delay with higher main memory consumption: instead of reclaiming memory upon commit, the primary cohort keeps log records of a completed transaction in memory until it finally receives an asynchronous acknowledgment from the backup. This aggressive version will be one of our future directions.

4.3 Failure recovery strategy

The primary-backup staggering provides redundancy for data stored in AutoDisk, which makes it resilient to various kinds of failures. In practice, AutoDisk can continue normal processing in the event of failures, as long as no (primary, backup) pair fail together. While this condition is met, both the data and the log of the active transaction on the failed PE are available, as the backup PE remains online and accessible, and the transaction can proceed without being suspended to wait for the failed PE to recover. This paper deals with failure scenarios where no logically adjacent PEs fail, i.e., either the primary or the backup remains operational. Failures at logically adjacent PEs cause both the primary and backup copy of some data to become inaccessible to the transaction, which requires the transaction to suspend until they recover. They will be dealt with in future research.

- Master failures

Like cohorts, a master also has a backup that is its logically neighboring PE. Failure in the backup master has no impact on the transaction processing. The primary master continues with its normal processing, except that the asyncnWAL register log is in its local main memory instead of in the memories of both the primary and backup PEs. If the primary master fails before it makes a decision about the transaction, the backup master will abort the transaction by sending the "abort" decision to all cohorts recorded in the membership log. If the primary master fails after making a decision but before receiving ACKs from all cohorts, the backup master will assume the master's task. Because the backup master has already received the membership log and the decision log from nWAL, it can rerun the decision phase by retransmitting the decision to all cohorts.

- Cohort failures

In the initial stage of a transaction, a master directly communicates with a primary cohort. A backup cohort failure during the transaction processing does not affect the master. In fact, the primary cohort doe not become aware of the backup cohort's failure before the decision phase, because there is no message synchronization between the primary and backup cohorts before the decision phase. The primary cohort detects the backup cohort's failure by timeout in the pending state waiting for an ACK from the backup cohort to acknowledge the completion of the transaction at the backup. After detecting a backup failure, the primary cohort continues normal processing by issuing an ACK to the master. At this point, the primary cohort retains all log records related to this transaction in main memory instead of deleting them to reclaim memory space, and passes them to the backup cohort when it starts a restart procedure. The backup cohort, on the other hand, can bring the data up to date with these log records received from the primary cohort. Handling failures at the primary cohort can be much more complex. The primary cohort may fail (1) before the final round of operations, which results in pending acknowledgments of operation for an active transaction, or (2) in the last prepared state before the master reaches a decision, or (3) in the pending state, which prevents the cohort from acknowledging the decision from the master. In case 1, the master can detect this failure by a timeout while waiting for an acknowledgment of operations. The master decides to abort the transaction, and delivers this decision to all other cohorts, including the backup cohort of the failed primary cohort. Cases 2 and 3 are directly related to commit processing. In these situations, the master has collected acknowledgments with implicit votes from all cohorts, reached a decision, and broadcast it to all primary cohorts. Failures in this circumstance are detected by timeout while waiting for an ACK of the decision. At this stage, operations of a transaction have already been successfully completed in all cohorts, so instead of aborting, the master resends the decision to backup cohorts. A decision message received by backup cohorts directly from the master informs them that the primary cohort is experiencing a failure. The backup cohort transmits an ACK to the master after it has applied the decision to the transaction. As a measure to help the primary's restart recovery, the backup cohort leaves the log records of that transaction intact in its memory instead of deleting them. During the restart recovery procedure, the primary cohort will ask the backup for a complete list of log records for the transaction in which it failed.

4.4 Protocol overheads comparison

Overheads referred to in this paper include message exchanges in both transaction processing and commit processing, as well as forced disk writes over the span of a transaction. Here we compute the overheads of BA-1.5PC analytically in comparison with two other protocols, 2PC and Early Prepare (EP).

Suppose there are N cohorts in a transaction besides the

Table 1. Overhead for committed transactions

Protocol	Number of messages	Number of forced log writes
BA-1.5PC	3+3P+4N	0
2PC	2P+8N	4N+1
EP	4P+2N	2P+2

master, and a transaction has P operations that are delivered to cohorts. We can discover from Figure 4 that BA-1.5PC requires a total of 3+3P+4N messages for the transaction. With the async-nWAL, by which a log write is converted into a message, BA-1.5PC in practice requires no forced disk writes of log records. The only disk write happens when updated data are moved to a disk asynchronously with the transaction, which is not counted in the overhead.

The basic 2PC requires a cohort to force-write two log records at commit (one for the vote and one for the decision). The master, on the other hand, must force-write a log record for the decision. The master also exchanges two rounds of messages with each cohort. With the messages to deliver the P operations to the cohorts, the total number of message exchanges reaches 4N+P, and there are 2N+1 forced disk writes. When extended to AutoDisk, the primary cohort must forward messages (about normal operation and commit processing) to its backup cohort, and the backup cohort will reply to these messages as well as force-write two log records during commit processing. The total number of messages therefore amounts to 8N+2P, with 4N+1 log writes.

EP requires a cohort to force-write its log record together with its updated data to stable storage on every operation, and to acknowledge that operation with a message to the master. In the commit phase, a cohort writes a forced log write only for aborted transactions, and sends acknowledgment only for aborted transactions. The master must force-write a membership log record at the beginning of a transaction and a decision log record in the commit phase. In this paper we are focused on committed transactions, and failure is a rather rare case in real systems, so we only count the overhead of committed transactions. The total overheads for such transactions are 2P+N messages and P+2 forced log writes. Similar to the overhead increase of 2PC when extended to AutoDisk, the overheads of cohorts double, because a primary cohort must transmit its operations and the final decision of a transaction to the backup cohort. The total overheads are 4P+2N messages and 2P+2 forced log writes for EP with AutoDisk.

Table 1 summarizes the overheads for the three protocols for committed transactions. Clearly, BA-1.5PC outperforms the other two, given that it does not require any forced log writes during a transaction. The relative ranking of 2PC and EP depends heavily on N and P. For Fat-Btree transactions where SMOs happen on index pages, the operation number P is u times N, where u is the number of updates to index pages, including insertion, splitting, and new allocation. Usually u is larger than two in such a SMO, therefore we can expect that EP incurs a larger overhead than 2PC arising from its heavy delays for forced log writes during operation processing.

5 Experiments and discussion

In this section, we evaluate the performance of the BA-1.5PC on a real running system. To evaluate the relative efficiency of our protocol, two other commit protocols, basic 2PC and Early Prepare (EP), are used for comparison. These two protocols were chosen because they are relatively suitable to be applied to AutoDisk by direct extension. 2PC is a classical protocol that can be applied to a broad category of databases. The basic form of 2PC, instead of a variant such as PC/PA, is used because they share the same structure and the variants differ from basic 2PC only by presumptions about certain transactions. EP was selected as a representative of 1-phase commit protocols. Other protocols that belong to the 1-phase commit category, such as Coordinator Log (CL) and Implicit Yes Vote (IYV) are excluded from our evaluation and experiments because of their basic policy that cohorts concentrate all or part of their log to the master, which effectively hampers the scalability of a distributed storage system that consists of many nodes.

We implemented BA-1.5PC, 2PC and EP on our experimental autonomous disks. Table 2 gives the basic configuration of this experimental system. Each node is regarded as an independent PE. The AutoDisk system consists of up to 64 PEs as storage nodes. Another eight PEs act as clients from which update requests are issued. Data in the AutoDisk system are indexed by a Fat-Btree parallel directory. First, objects are inserted into the AutoDisk system. An object is treated as a key with a fixed amount of data (currently 400 bytes), and all objects are distributed evenly into storage nodes. The clients then issue insertion requests randomly to storage nodes in AutoDisk to measure the throughput of insertion transactions. This testing was repeated for different sizes of dataset (different numbers of objects) and with varying numbers of storage nodes in the AutoDisk system.

Figure 5 presents the insertion transaction throughput with BA-1.5PC, 2PC and EP, for several dataset sizes in the AutoDisk system with 64 storage nodes. The experimental results indicate that BA-1.5PC outperforms 2PC and EP for transaction throughput at all dataset sizes. It is not surprising that BA-1.5PC gives overwhelmingly superior transaction throughput over 2PC and EP, because BA-

Table 2. Configuration of experimental system

Nodes	64 Storage nodes, 8 client nodes
CPU	AMD Athlon XP-M1800+ (1.53 GHz)
Memory	PC2100 DDR 1 GB
Disks	Toshiba MK3019GAX (30 GB, 5400 rpm, 2.5 inch)
OS	Linux 2.4.20
Java VM	Sun J2SE 1.4.2_04 Server VM

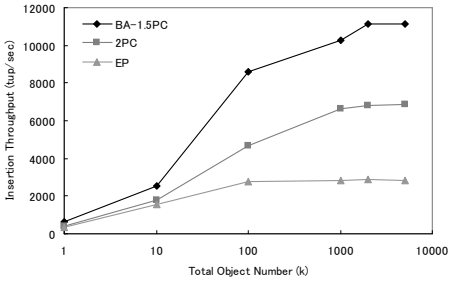

Figure 5. Comparison of throughput with BA-1.5PC, 2PC and EP with 64 nodes

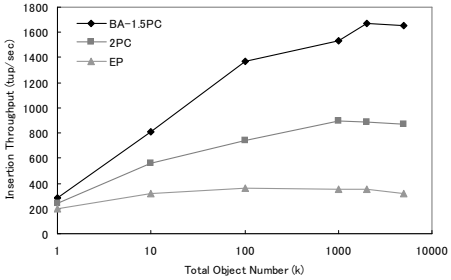

Figure 6. Throughputs with eight nodes

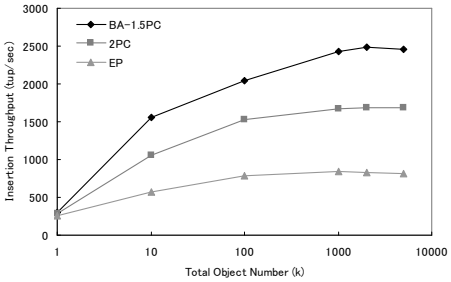

Figure 7. Throughputs with 16 nodes

1.5PC greatly reduces the message complexity during the commit process, and in practice eliminates the necessity to write forced disk log records. Early Prepare (EP), on the other hand, produces a less effective performance than 2PC even though it requires only one phase to commit a transaction instead of two. Although EP uses one less phase in the commit process, it requires every cohort to force-write a log record for every operation during a transaction. It is this conservative disk log write that diminishes its throughput.

Figures 6, 7 and 8 give the insertion transaction throughput for different numbers of storage nodes in the AutoDisk system for BA-1.5PC, 2PC and EP, respectively. We can see that in general the transaction throughput increases monotonically as the dataset in AutoDisk grows larger. This can be ascribed to the Fat-Btree index structure. More objects in a PE make it less likely that insertions of an object will be directed to a certain leaf page. It is then less likely that a leaf page will overflow to result in an expensive index page split, which in turn causes an expensive Fat-Btree index page update. A larger dataset is therefore concomitant with a higher transaction throughput.

When the number of objects exceeds some critical amount, it becomes impossible to keep all the index pages within the cache memory because of the limited cache buffer. In this situation, retrieval of an index page will probably encounter a blocking disk access and incur a heavy delay. We can observe that the throughput ceases to increase and in fact drops slightly for dataset sizes of greater than two million objects. We can also find from the figures that more storage nodes in the system allow higher insertion throughputs. This is attributed to the good scalability and parallelism of the Fat-Btree index structure.

6 Conclusions and future work

We have proposed a new commit protocol, BA-1.5PC, which addresses the problem of providing efficient transaction commit processing even in the face of failures. The protocol also surmounts the problem that cohorts may be blocked by a master failure in the decision phase. It yields these merits by combining the following strategies. (1) A low-overhead log mechanism based on an asynchronous version of neighbor WAL (async-nWAL), which greatly reduces the need for disk I/O and eliminates a synchronization message from nWAL to reduce message overhead. (2) Removal of the voting phase from commit processing to gain a faster commit process. The votes of cohorts are included implicitly in the acknowledgment messages to the master during transaction processing. (3) A primary-backup assisted recovery strategy to enhance dependability in the face of possible failures. Failures of cohorts can be recovered

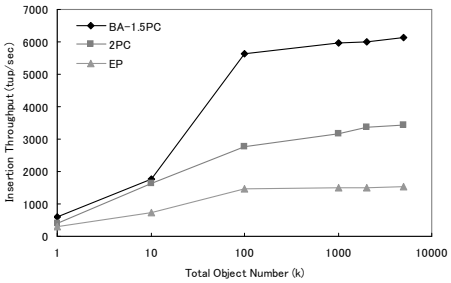

Figure 8. Throughputs with 32 nodes

easily with assistance from the backup cohorts. In addition, a master's failure in the decision phase will not block other cohorts with this strategy. Prepared cohorts can receive a decision from a master's backup after such a failure.

We implemented BA-1.5PC in an experimental AutoDisk system and ran experiments to evaluate its effectiveness. Two other commit protocols, 2PC and EP, were included in our experiment for comparison with BA-1.5PC. The results prove that BA-1.5PC enjoys a superior performance advantage in terms of transaction throughput for varied dataset sizes and at different AutoDisk system scales.

Our experiments covered only the normal case, with no failures during the transaction process. Testing of transactions with failures and the failure recovery strategy will be conducted in the near future.

Acknowledgments

This research is sponsored by CREST of JST (Japan Science and Technology Agency), MEXT of the Japanese Government, a Grant-in-Aid for Scientific Research on Priority Areas (#16016232, #18049026), the SRC, the Tokyo Institute of Technology 21COE Program "Framework for Systematization and Application of Large-Scale Knowledge Resources", and the NHK Science and Technical Research Laboratories.

References

[1] M. Abdallah, R. Guerraoui, and P. Pucheral. Dictatorial transaction processing: Atomic commitment without veto right. *Distributed and Parallel Databases*, 11(3):239–268, 2002.

[2] Y. Al-Houmaily and P. Chrysanthis. Two-phase commit in gigabit-networked distributed databases. In *Proc. of 8th International Conference on Parallel and Distributed Computing Systems*, Sep. 1995.

[3] O. Babaoglu and S. Toueg. Understanding non-blocking atomic commitment. *Distributed Systems, Addison Wesley*, 1993.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] C. Bobineau, P. Pucheral, and M. Abdallah. A unilateral commit protocol for mobile and disconnected computing. In *Proc. Of International Conference on Parallel and Distributed Computing Systems*, 2000.

[6] J. Gray. A comparison of the byzantine agreement problem and the transaction commit problem. In *Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, pages 10–17. Springer, 1986.

[7] R. Gupta, J. R. Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *Proc. of SIGMOD Conference*, pages 486–497, 1997.

[8] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[9] H.-I. Hsiao and D. J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proc. of ICDE'90*, pages 456–465, 1990.

[10] S.-O. Hvasshovd. Recovery in parallel database systems. *Database Systems, Vieweg*, 1996.

[11] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and S. Arévalo. A low-latency non-blocking commit service. In *Proc. of DISC2001*, pages 93–107, 2001.

[12] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Record*, 27(3):42–52, 1998.

[13] J. Miyazaki and H. Yokota. Concurrency control and performance evaluation of parallel b-tree structures. *IEICE Transactions on Information and Systems*, E85-D(8):1269–1283, Aug. 2002.

[14] C. Mohan, B. G. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[15] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. of VLDB'98*, pages 62–73, 1998.

[16] D. Skeen. *Crash Recovery in a Distributed Database Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1982.

[17] D. Skeen. A quorum-based commit protocol. In *Proc. of Berkeley Workshop*, pages 69–80, 1982.

[18] J. W. Stamos and F. Cristian. A low-cost atomic commit protocol. In *Proc. of SRDS'90*, pages 66–75, 1990.

[19] J. W. Stamos and F. Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4):383–408, 1993.

[20] H. Yokota. Autonomous disks for advanced database applications. In *Proc. of DANTE'99*, pages 435–442, 1999.

[21] H. Yokota, Y. Kanemasa, and J. Miyazaki. Fat-btree: An update-conscious parallel directory structure. In *Proc. of ICDE'99*, pages 448–457, 1999.

[22] T. Yoshihara, D. Kobayashi, R. Taguchi, and H. Yokota. A concurrency control method for parallel btree structures. In *Proc. of SWOD2006, in conjunction with ICDE2006*, pages 71–76, 2006.