Tolerating Byzantine Faults in Transaction Processing Systems using Commit Barrier Scheduling

Ben Vandiver, Hari Balakrishnan, Barbara Liskov, Sam Madden
MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA, USA
{benmv,hari,liskov,madden}@csail.mit.edu

ABSTRACT

This paper describes the design, implementation, and evaluation of a replication scheme to handle Byzantine faults in transaction processing database systems. The scheme compares answers from queries and updates on multiple replicas which are unmodified, off-the-shelf systems, to provide a single database that is Byzantine fault tolerant. The scheme works when the replicas are homogeneous, but it also allows heterogeneous replication in which replicas come from different vendors. Heterogeneous replicas reduce the impact of bugs and security compromises because they are implemented independently and are thus less likely to suffer correlated failures.

The main challenge in designing a replication scheme for transaction processing systems is ensuring that the different replicas execute transactions in equivalent serial orders while allowing a high degree of concurrency. Our scheme meets this goal using a novel concurrency control protocol, *commit barrier scheduling* (CBS). We have implemented CBS in the context of a replicated SQL database, HRDB (Heterogeneous Replicated DB), which has been tested with unmodified production versions of several commercial and open source databases as replicas. Our experiments show an HRDB configuration that can tolerate one faulty replica has only a modest performance overhead (about 17% for the TPC-C benchmark). HRDB successfully masks several Byzantine faults observed in practice and we have used it to find a new bug in MySQL.

Categories and Subject Descriptors: D.4.5 [**Reliability**]: Fault-tolerance; H.2.4 [**Systems**]: Concurrency

General Terms: Design, Performance, Reliability

1. INTRODUCTION

Transaction processing database systems are complex, sophisticated software systems involving millions of lines of code. They need to reliably implement "ACID" semantics, while achieving high transactional throughput and high availability. As is usual with systems of this size, we can expect them to contain thousands of fault-inducing bugs in spite of the effort in testing and quality assurance on the part of vendors and developers.

A bug in a transaction processing system may immediately cause a crash; if that happens, the system can take advantage of its transactional semantics to recover from the write-ahead log and the only impact on clients is some downtime during recovery. However, bugs may also cause *Byzantine faults* in which the execution of a query is incorrect, yet the transaction commits, causing wrong answers to be returned to the client or wrong data items to be stored in the database. Examples of such faults include concurrency control errors, incorrect query execution, database table or index corruption, and so on (see Section 6). In fact, even if a bug eventually results in a crash, the system could have performed erroneous operations (and exhibited Byzantine behavior) between the original occurrence of the bug and the eventual crash. Such faulty behavior is hard to mask because it is difficult to tell if any given statement was executed correctly or not. Existing database systems offer no protection against such faults.

This paper describes the design, implementation, and evaluation of a new replication scheme to handle both Byzantine and crash faults in transaction processing systems. Our approach can be used with any system that supports transactions consisting of "read" and "write" operations (*e.g.*, read and update queries) followed by a COMMIT or ABORT. Thus it can be used with relational database systems, object-oriented databases and object stores, semi-structured (XML) query processing systems, etc. Our scheme requires *no modifications to any database replica software*; in fact, it does not require any additional software to run on any machine hosting a replica database. Treating each replica as a "shrink-wrapped" subsystem eases the deployment and operation of our system and allows it to work with commercial offerings.

The primary goal of our system is correctness. We must provide a *single-copy serializable view* of the database: the group of replicas must together act as a single copy, assuming no more than some threshold number of replicas, f, are faulty. However, we also require that the performance of our approach must not be substantially worse than that of a single database. Achieving both goals simultaneously is challenging because databases achieve high performance through concurrency. The problem is that, if presented with a workload consisting of operations from a set of concurrent transactions, different replicas may execute them in different orders, each of which constitutes a correct serial execution at that replica. However, these local orders may not all be consistent with a single global order, which is needed for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010 ...$5.00.

correctness. Ordering problems can be avoided by running one transaction at a time, but this approach eliminates all concurrency and performs poorly.

The key contribution of this paper is a new concurrency control protocol, called *commit barrier scheduling* (CBS), that allows our system to guarantee correct behavior while achieving high concurrency. CBS constrains the order in which queries are sent to replicas just enough to prevent conflicting schedules, while preserving most of the concurrency in the workload. Additionally CBS ensures that users see only correct responses for transactions that commit, even when some of the replicas are Byzantine faulty. To our knowledge, CBS is the first approach that provides both correctness and good performance for replicated transaction processing systems that handle Byzantine failures. CBS requires that each replica implement concurrency control using strict two-phase locking, but this is not onerous since two-phase locking is used in many production databases.

Our system can be deployed with an identical set of replicas, but we also support two other deployment scenarios. First, the replicas might run different versions of a database coming from a single vendor. Such a configuration provides a way to test new versions, as well as a way to release new versions without exposing clients to effects of introduced errors. Second, the replicas might come from different vendors. Such heterogeneous replicas are unlikely to share the same set of bugs since their implementations were produced independently. Heterogeneity can also reduce the probability that security vulnerabilities will affect the correct behavior of the replicated system, because a single vulnerability will likely affect only one of the replicas. Clearly all replicas in a deployment must be similar, *e.g.*, all are SQL relational databases. Furthermore all non-faulty replicas must process queries "identically"; we discuss this requirement further in Section 4.

We have implemented CBS as part of *HRDB (Heterogeneous Replicated DataBase)*, which uses SQL databases as replicas. HRDB supports replicas from different vendors (we have used IBM DB2, MySQL, Microsoft SQLServer, and Derby). Our experiments with the HRDB prototype show that it can provide fault-tolerance by masking bugs that exist in some but not all replicas. HRDB is capable of masking deterministic bugs using replicas from heterogeneous vendors and non-deterministic bugs using different versions from the same vendor. In addition, using HRDB we discovered a serious new non-deterministic bug in MySQL; a patch for this bug will be included in the next MySQL release.

We found HRDB to have reasonable overhead of about 17% (compared to a single replica) on TPC-C, a database industry standard benchmark that models a high-concurrency transaction processing workload. Analysis of this overhead reveals that much of it is processing overhead, indicating that even better performance can be achieved with further tuning.

The rest of this paper is organized as follows. We begin in Section 2 by discussing related work. Section 3 describes the design of HRDB, commit barrier scheduling, and how our scheme handles faults. Section 4 presents some practical issues related to database replication, while Section 5 describes our implementation. Sections 6 presents a survey of bugs and our success at masking them with HRDB. We demonstrate reasonable performance in Section 7 and conclude in Section 8.

2. RELATED WORK

To the best of our knowledge, HRDB is the first practical Byzantine fault-tolerant transaction processing system that is able to execute transactions concurrently. We begin by discussing work on the use of replication to allow database systems to survive crashes. Then we discuss research on systems that tolerate Byzantine faults.

2.1 Database Replication for Crash Faults

Almost all work on database replication has dealt only with crash failures. Several research projects and commercial products use replication to avoid the downtime caused by crashes [3, 4, 12, 14, 9, 5]. Systems can typically be classified into eager or lazy replication based on whether they propagate updates inside transaction boundaries or not. They also differ in whether the clients observe a single-copy serializable view [14, 5, 4] or a more lax concurrency model [20, 12, 10]). Some work also explores the problem of providing availability in a partitioned system [1]. Replicated DBMSs also generally require access to the write-sets of each of the transactions

None of these approaches can be used to tolerate Byzantine faults, because all these systems assume that the primary replica processes the queries correctly. Its results are simply propagated to the other secondary replicas without any checking. For example, some systems use log shipping or write-set extraction as a way to communicate results from the primary to the secondaries [11, 24]. This approach cannot provide Byzantine fault tolerance unless the log contains the queries themselves, since otherwise it isn't possible for the secondaries to check what the primary did. Furthermore, the answers returned to the client do not necessarily appear in the log, and yet they must also be checked. Our approach overcomes these problems while still providing good performance.

Some work on replication for databases resembles ours in that it requires no changes to the replicas; instead these systems have been implemented as "middleware" to separate them from the complexity of DBMS internals [18, 21, 16, 8]. None of these systems tolerate Byzantine faults. Clients interact with the middleware, which distributes work to the database replicas to provide fault-tolerance or scalability. These systems require a concurrency control scheme to keep the replicas synchronized so that they can provide some transaction isolation guarantee to the user. One approach requires clients to pre-declare the tables involved in the query, and then schedules transactions using coarse-grained (table-granularity) locks [2]. Rather than imposing a schedule, our approach uses the databases' own concurrency manager to avoid introducing artificial dependencies, thus permitting higher concurrency. The Pronto [19] system for high availability uses a primary-secondary scheme for transaction ordering, but executes transactions serially on the secondaries. C-JDBC [8] is similar to our approach in performing replication entirely on the SQL level and supporting heterogeneous databases, but it runs updates and commits sequentially. Finally, some recent work has focused on providing *snapshot isolation* (SI) instead of serializability [21, 16] to take advantage of SI implementations in PostgreSQL and Oracle. These systems preserve some concurrency in a middleware-based replication system, but use knowledge of the write-set of transactions to resolve ordering conflicts.

2.2 Tolerating Byzantine Faults

A scheme for Byzantine fault-tolerant database replication was proposed two decades ago [17], but this work does not appear to have been implemented and the proposal did not exhibit any concurrency, implying that it would have performed quite poorly.

Gashi et al. [10] discuss the problem of tolerating Byzantine faults in databases and document a number of Byzantine failures in bug reports from real databases (see the discussion in Section 6). They propose a middleware-based solution where transactions are issued to replicas and the results are voted on to detect faults. They do not, however, present a protocol that preserves concurrency or discuss the associated problem of ensuring equivalent serial schedules on the replicas.

The BFT library [6] provides efficient Byzantine fault-tolerance through state machine replication; it requires that all operations be deterministic. One might ask whether this library, previously used for NFS, can be easily adapted to transactional databases. BFT ensures that all the replicas process operation requests in the same order; in our system operations are queries, COMMITs, and ABORTs. As originally proposed, BFT requires that operations complete in order; with this constraint, the first query to be blocked by the concurrency control mechanism of the database would cause the entire system to block. The authors of [15] propose a way to loosen this constraint by partitioning operations into non-conflicting sets and running each set in parallel. To do this, however, they require the ability to determine in advance of execution whether two operations (or transactions) conflict, which is possible in some systems (such as NFS). In database systems, however, the statements of the transaction are often not available at the time the transaction begins. Furthermore, just determining the objects that a single database statement will update involves evaluating predicates over the contents of the database (an operation that must itself be properly serialized.) To see why, consider an update in an employee database that gives a raise to all employees who make less than $50,000; clearly, it is not possible to determine the records that will be modified without (partially) evaluating the query. In contrast, our scheme does not require the ability to determine the transaction conflict graph up front but still preserves concurrency while ensuring that state machine replication works properly.

BASE [7] is an extension of BFT that allows the use of heterogeneous replicas by implementing stateful conformance wrappers for each replica. Typically, the conformance wrapper executes the operations while maintaining additional state that allows it to translate between the local state of the replica and the global system state. Our system interacts with replicas via SQL, which is a more expressive interface than the BASE examples. Rather than maintaining state in the wrapper, our scheme depends on rewriting the SQL operations to keep the replicas' logical state the same. A more full featured implementation might maintain wrapper state to circumvent some vagaries of SQL.

Any system that tolerates Byzantine faults requires that no more than 1/3 of the replicas are faulty. Recent work [26] shows that agreement under a Byzantine fault model with f allowed faulty nodes requires $3f + 1$ replicas, but execution only requires $2f + 1$. Thus a two-tier system can use $3f + 1$ lightweight nodes to perform agreement and only $2f + 1$ nodes to execute the workload. We use this property in our design.

3. HRDB DESIGN

This section describes our system. We begin by discussing our assumptions about databases. Then we provide an overview of our approach. Sections 3.3 through 3.5 present the details. We conclude in Section 3.6 with a discussion of the correctness of our system.

3.1 Database Assumptions

Each database transaction consists of a sequence of *queries*, followed by a COMMIT or ABORT. Queries can read, update, insert, or delete entries in the database. Database systems ensure that queries are executed at the database in the order they are issued by the client. We assume that clients wait for the response to one statement before sending the next.[1]

Databases provide ACID semantics for transactions [13], which requires both a concurrency control model and a recovery mechanism. For the former, we require serializable execution: the effect of running a group of transactions is the same as running them sequentially in some order. The latter typically takes the form of a write-ahead log that is used by the system to recover from crash failures; after recovery the system guarantees that modifications of all transactions that committed survive the crash, and that modifications of all other transactions are discarded. We assume the existence of a recovery mechanism (all production databases provide such).

While several different concurrency control mechanisms have been proposed, we require the databases in our system to use *strict two-phase locking*. With strict two-phase locking transactions acquire read and write locks on data items as they access them, and these locks are held until the end of the transaction. Strict two-phase locking is the most commonly used concurrency control mechanism; for example, it is used in SQLServer, MySQL, and DB2. We cannot handle databases (like Oracle) that use alternative techniques like *snapshot isolation*, which do not provide true serializability. In ongoing work, we are looking at ways to relax our concurrency control requirement.

Concurrency control mechanisms such as two-phase locking determine the synchronization constraints as transactions run, by observing the items they access. Thus the serial schedule is determined after the fact. It is sometimes possible, given a limited workload, to preprocess queries and determine the schedule in advance, but this is very difficult to do in general. Since we want our system to work for an arbitrary mix of transactions, we do not rely on such preprocessing.

3.2 Overview

In HRDB, clients do not interact directly with the database replicas. Instead they communicate with a *shepherd*, which acts as a front-end to the replicas and coordinates them. Figure 1 shows the HRDB system architecture. Client applications do not know they are talking to a replicated system because HRDB provides a single-copy serializable view of the replicated database. We assume that all

[1]This assumption simplifies exposition. The system also works when multiple statements of a transaction are presented in a single request.

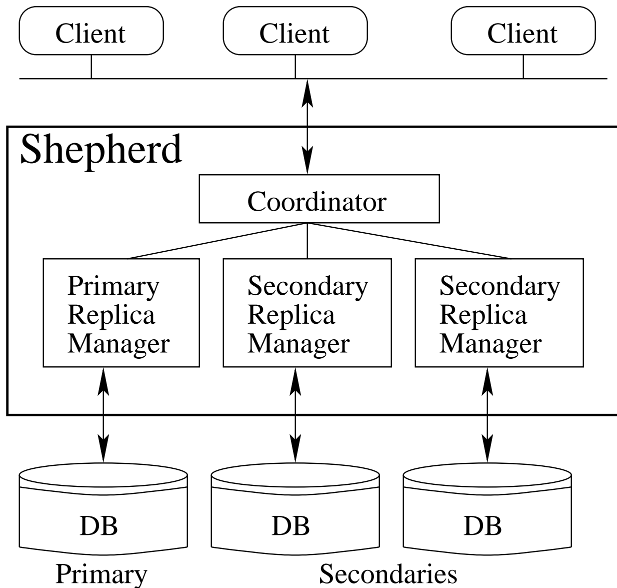

Figure 1: HRDB system architecture.

transactions that run on the replicas are sent via the shepherd.

HRDB is parametrized by f, the number of simultaneously faulty replicas it can tolerate. It requires only $2f+1$ database replicas because the replicas do not carry out agreement; instead they simply execute statements sent to them by the shepherd. The figure illustrates a system in which $f = 1$.

In this paper, we assume the shepherd itself is trusted and does not have Byzantine faults, though it might crash (*e.g.*, because the machine it runs on crashes). Since the complexity and amount of code in the shepherd is orders of magnitude smaller than in the replicas, we believe that assuming non-Byzantine behavior is reasonable. Furthermore, we expect that a trusted shepherd is the most likely way the system would be deployed in practice. Nevertheless we are investigating (as future work) ways to make the shepherd Byzantine-fault tolerant by running $3f + 1$ shepherd replicas; the problem turns out to be non-trivial if we want to avoid significant performance degradation.

As shown in the figure, the shepherd runs a single *coordinator* and one *replica manager* for each back-end replica. The coordinator receives statements from clients and forwards them to the replica managers. Replica managers execute statements on their replicas, and send answers back to the coordinator. The coordinator sends results back to the clients, compares query answers for agreement, and determines when it is safe for transactions to commit. The coordinator may also decide to abort and retry transactions or initiate repair of faulty replicas.

The coordinator and replica managers must run the statements of transactions in a way that satisfies the following three objectives:

- Ensure that all non-faulty replicas have equivalent logical state such that they will return the same answer to any given query.
- Ensure that the client always gets correct answers to queries belonging to transactions that commit, even when up to f replicas are faulty.
- Detect faulty replicas and flag them for repair.

The first objective can be met by coordinating the replicas to guarantee that they all execute equivalent serial schedules. The naive approach of running transactions one at a time to effect equivalent serial schedules on the replicas has poor performance. At the other extreme, simply sending the transactions concurrently to a set of replicas will usually lead to different serial schedules, because each replica can pick a different, yet "correct", serial ordering. Our solution to this problem is *commit barrier scheduling (CBS)*. CBS ensures that all non-faulty replicas execute committed transactions in equivalent serial orders, while at the same time preserving much of the concurrency of the individual replicas.

In CBS, one replica is designated to be the *primary*, and runs statements of transactions slightly in advance of the other *secondary* replicas. The order in which transactions complete on the primary determines a serial order. CBS ensures that all the non-faulty secondaries commit transactions in an order equivalent to that at the primary. To improve performance, the shepherd observes which queries from different transactions are able to execute concurrently without conflicts at the primary, and allows the secondaries to execute these queries concurrently.

CBS depends on our assumption that the replicas use strict two-phase locking for two reasons. First, strict two-phase locking exposes information about concurrency available in the workload, which we can use to achieve good performance. Strict two-phase locking ensures that a transaction T will hold all read and write locks acquired by any query in T until after T commits (or aborts). With two-phase locking if a transaction T_1 is able to complete a query Q_1 after a query Q_2 from transaction T_2 has run (or is running) but before T_2 commits (or aborts), then Q_1 and Q_2 do not conflict and can be executed in any order. Furthermore, Q_1 does not conflict with any statement of T_2 before Q_2. The coordinator uses this property to extract concurrency information by observing the primary execute transactions: if the (non-faulty) primary allows a set of queries to run without conflicting, then the secondaries can run these queries concurrently (*e.g.*, in parallel), or in any order, without yielding a different equivalent serial ordering from the primary.

The second reason CBS depends on strict two-phase locking has to do with the correctness of our voting mechanism. If a transaction has executed to completion and produced some answers, the shepherd must have confidence that the transaction could actually commit with those answers. Under strict two-phase locking, a transaction holds locks until the end of the transaction; thus a transaction that has executed all of its queries will be able to commit (unless the application issued an ABORT or the system crashed), because it has all the resources it needs to do so. Not all concurrency control protocols satisfy these two properties, which is why we do not support them.

Our approach of using the primary to determine if queries can run in parallel performs well only when the primary's locking mechanism is an accurate predictor of the secondary's mechanism. For example if the primary locks individual rows, while the secondary locks pages, the secondary might block when the primary does not. Such blocking is

not a correctness issue, but it does affect performance. To avoid it we additionally require (for good performance) that concurrency control at the replica selected to be the primary is *sufficiently blocking*:

> A replica is *sufficiently blocking* if, whenever it allows two queries to run in parallel, so do all other non-faulty replicas.

For good performance CBS requires $f + 1$ sufficiently blocking replicas so that we can freely use any of them as a primary

3.3 Commit Barrier Scheduling

CBS does not limit concurrency for processing queries at the primary in any way. When the coordinator receives a query from a client, it immediately sends it to the primary replica manager, which forwards it to the primary replica. Hence, the primary replica can process queries from many transactions simultaneously using its internal concurrency control mechanism (strict two-phase locking). As soon as it returns a response to a query, the coordinator sends that query to each secondary replica manager. Each of them adds the query to a pool of statements that will eventually be executed at the corresponding secondary.

The pool of statements for a secondary manager can contain many statements, and these statements can come from multiple transactions. The job of the secondary manager is to send these statements to the secondary, with as much concurrency as possible, while ensuring that the secondary, in the absence of faults, will execute the statements in a serial order equivalent to that at the primary. To achieve this property, the secondary manager *delays* sending statements to the secondary replica, using three ordering rules:

- *Query-ordering rule.* A query or COMMIT of transaction T can be sent to the secondary only after the secondary has processed all earlier queries of T.
- *Commit-ordering rule.* A COMMIT for transaction T can be sent to a secondary only after the secondary has processed all queries of all transactions ordered before T.
- *Transaction-ordering rule.* A query from transaction T_2 that was executed by the primary after the COMMIT of transaction T_1 can be sent to a secondary only after it has processed all queries of T_1.

These rules are the only constraints on delaying statement execution on the secondaries; they permit considerable concurrency at the secondaries.

The first two rules are needed for correctness. The query-ordering rule ensures that each individual transaction is executed properly. The commit-ordering rule ensures that secondaries serialize transactions in the same order as the primary.

The transaction-ordering rule is needed only for performance because it avoids deadlocks at secondaries. For example, suppose query Q_2 of T_2 ran at the primary after the primary committed T_1. If a secondary ran Q_2 before running some statement Q_1 from T_1, Q_2 might acquire a lock needed to run Q_1, thus preventing T_1 from committing at the secondary. The transaction-ordering rule prevents this reordering. However, it allows statements from T_2 that ran before T_1 committed on the primary to be run concurrently with statements from T_1 on the secondaries (such statements are guaranteed not to conflict because strict two-phase locking would not have allowed Q_2 to complete on the primary if it had any conflicts with Q_1.) Hence, CBS preserves concurrency on the secondaries. In particular, if the secondary pool contains a number of queries, each from a different transaction, and there are no COMMITs in the pool, all of the queries can be sent to the secondary replica concurrently.

The coordinator sends the primary's response to a query to the client as soon as it receives it. If the primary is faulty, this response may not be correct. It may seem that we could avoid sending incorrect responses to the client by waiting until f secondaries provide results that agree with that of the primary, but waiting does not always work when the primary is Byzantine-faulty, as explained in section 3.3.2.

Instead, we determine the correctness of the response after the fact and abort the transaction if the response is incorrect. The coordinator delays processing a COMMIT request for a transaction T until at least $f+1$ replicas are *ready to commit* T:

> A replica is *ready to commit* transaction T if it has processed all queries of T and also all queries of every transaction ordered before T.

The coordinator will allow the commit only if the query results sent to the client are each agreed to by f secondaries that are ready to commit T; otherwise it aborts the transaction. Thus we ensure that the client receives correct answers for all transactions that commit.

3.3.1 Commit Barriers

We implement the transaction-ordering rule using *commit barriers*. The coordinator maintains a global commit barrier counter, B. Before the coordinator sends a COMMIT request for transaction T to the replicas, it sets T's barrier, $T.b$, to B, then increments B. Additionally, when the coordinator gets a response to query Q from the primary, it sets Q's barrier, $Q.b$, to B. A secondary replica manager waits to send Q to the secondary until it has received responses from all queries of committed transactions T such that $T.b \leq Q.b$.

The use of commit barriers is conservative: it may delay a query unnecessarily (*e.g.*, when the query doesn't conflict with the transaction whose COMMIT is about to be processed). It is correct, however, because delaying the running of a query isn't wrong; all it does is cause the processing of that transaction to occur more slowly.

The pseudo-code for the coordinator is given in Figure 2. The code is written in an event-driven form for clarity, although our implementation is multi-threaded. The code for the primary replica manager is not shown; this manager is very simple since it sends each statement to its replica as soon as it receives it from the coordinator, and returns the result to the coordinator. Figure 3 shows the code for a secondary replica manager.

Figure 4 shows an example schedule of three transactions, T1, T2, and T3, executed by the primary. Each COMMIT ("C") causes the barrier, B, to be incremented. With CBS, the secondary replicas can execute the statements from different transactions in the same barrier (*i.e.*, between two COMMITs) in whatever order they choose; the result will be equivalent to the primary's serial ordering. Of course, two statements from the same transaction must be executed in the order in which they appear. Note that CBS does not

- Event: Receive query Q from client.
  Action: Send Q to primary replica manager.
- Event: Receive response for query Q from the primary replica manager.
  Actions:
  1. Send the response to the client.
  2. $Q.b \leftarrow B$.
  3. Record response as $Q.ans$.
  4. Send Q to secondary replica managers.
- Event: Receive response from a secondary replica manager for query Q.
  Action: Add response to $votes(Q)$.
- Event: Receive ABORT from client.
  Actions:
  1. Send ABORT to replica managers.
  2. Send acknowledgment to client.
- Event: Receive COMMIT for transaction T from client.
  Actions: Delay processing the COMMIT until $f+1$ replicas are ready to commit T. Then:
  1. If the response $Q.ans$ sent to the client for some query Q in T is not backed up by f votes in $votes(Q)$ from replicas that are ready to commit T, send ABORT to the replica managers and inform the client of the ABORT.
  2. Otherwise:
     (a) $T.b \leftarrow B; B \leftarrow B+1$.
     (b) Send acknowledgment to client.
     (c) Send COMMIT to replica managers.

Figure 2: Pseudo-code for the coordinator.

1. For each ABORT statement for a transaction T in the pool, discard from the pool any queries of T that have not yet been sent to the replica and send the ABORT to the replica.
2. For each query Q in the pool, determine whether it is ready as follows:
   (a) All earlier queries from Q's transaction have completed processing.
   (b) All queries of committed transactions T such that $T.b \leq Q.b$ have completed processing.
3. A COMMIT of transaction T in the pool is ready if all queries such that $Q.b < T.b$ have completed processing at the secondary replica.
4. Execute each query Q or COMMIT that is ready on the replica and send the result to the coordinator.

Figure 3: Pseudo-code for secondary managers.

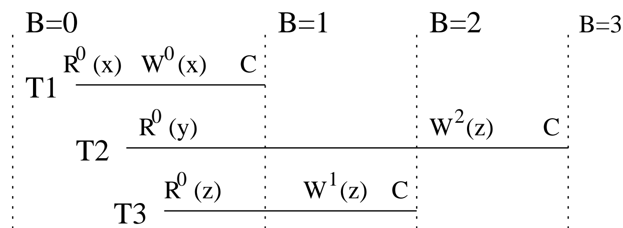

Figure 4: Example schedule of three transactions, as executed by the primary. Note that the coordinator does not know the identities of x, y, and z (or even that they are distinct). Each query is super-scripted with the barrier CBS would assign to it.

Faulty primary	Replica R1	Replica R2
$T_1 : A = 1$	$T_1 : A = 1$	T_1: Waiting for T_2
$T_2 : A = 1$	T_2: Waiting for T_1	$T_2 : A = 1$

Figure 5: A faulty primary's reply could match a non-faulty secondary's reply and be the majority. The variable A is initially 0 at all replicas, and each transaction increments the value of A by 1.

extract all available concurrency. For example, a secondary manager delays sending T3's $W(z)$ to the secondary until after the secondary has processed all queries of T1. This is because we assume, conservatively, that T3's $W(z)$ *might* conflict with queries of T1. The same rule prevents sending T2's $W(z)$ before T3's $W(z)$ has been processed and in this case the delay is needed since there is a conflict.

3.3.2 Handling a Faulty Primary

The pseudo-code in Figures 2 and 3 provides correct behavior even when the primary is Byzantine faulty. Obviously it handles a primary that fails by returning a wrong answer. However a faulty primary might have a concurrency control bug, so that it allows transactions that conflict to run in parallel. A concurrency error is most likely to lead to transactions being unable to make progress because they block at secondaries; we handles such cases by selective aborts and retries on the secondaries. But it is also possible that a transaction is able to make progress even when the primary has a concurrency control bug.

The example in Figure 5 illustrates the point. Consider two transactions, T_1 and T_2, each of which increments a variable A and returns its value. If A were 0 to start with, then the correct value after executing both transactions should be 2. However if the primary is faulty the following might occur: The primary runs T_1 and T_2 concurrently, incorrectly returning $A = 1$ as the result of both increment statements. Each secondary is correct; however, replica R1 runs T_1 before T_2, acquiring the lock for A and blocking T_2. Similarly, replica R2 runs T_2 before T_1, blocking T_1. Furthermore, replica R1 endorses the primary's response for T_1 and replica R2 endorses the primary's response for T_2. Thus waiting for $f+1$ replicas to agree on an answer before sending it to the client does not ensure that the answer is correct!

The scenario illustrates two issues that arise when the primary is Byzantine-faulty. The first is that non-faulty replicas can have trouble processing COMMITs. For example, suppose T_1 commits first. Then replica R2 will be unable

to process the COMMIT because it hasn't finished executing the statements of T_1. To handle this problem, a replica manager that is unable to process a COMMIT will abort other transactions that are "in progress", *i.e.*, for which the replica has processed some statements but has not yet received a COMMIT. Thus the replica manager for replica R2 will eventually abort T_2, allowing it to finish processing T_1. It can then re-run T_2.

The second issue is that when transactions re-run, they can produce different answers. When replica R2 aborts T_2, commits T_1, and then re-runs T_2, it will produce the final answer of 2. Replica R1 will also produce this answer, since it runs T_2 after the commit of T_1. Furthermore, this answer is correct and we need to notice that the answer sent to the client for T_2 was wrong and abort the transaction.

Our protocol ensures correct behavior because it requires matching query responses from replicas that are ready to commit transaction T. One point to note here is that a replica can change its vote and we consider only the last vote provided by the replica in deciding whether the transaction can commit. Once a replica is ready to commit a transaction, its answer is stable (*i.e.*, it will never change), assuming the replica is non-faulty.

3.3.3 Discussion

With CBS, clients must be prepared to occasionally receive wrong answers for transactions that abort. Thus, all answers in CBS must be considered "tentative" until the commit point, after which they are guaranteed to be correct (assuming no more than f faulty replicas). Our semantics will not cause a problem for clients that work in "auto commit" mode (COMMIT sent implicitly after each query). Our semantics will also not be a problem for clients that respect transactional semantics and do not let values produced by a transaction "escape" to the application prior to commit — most database applications are programmed in this way. Finally, note that even if bad results did escape, the situation is still better than it is today without CBS, because CBS will abort the transaction and ensure that the replicas have the correct logical state, whereas in existing systems, wrong answers are simply returned without any warning.

An additional point is that we get performance benefits from returning the primary's answer to the client immediately, as it reduces statement latency as observed by the client. It allows the processing of transactions to be pipelined: the primary executes the next statement while the secondaries execute the previous one. Sending the primary's response to the client as soon as it arrives also improves overall system performance because transactions complete more quickly and therefore hold database locks for shorter periods.

CBS allows a replica manager to send commits from different transactions to the replica concurrently. Having multiple outstanding commits allows the database to make use of group commit, an important optimization for reducing stable storage overhead.

A final point is that if the primary is not faulty and its concurrency control is sufficiently blocking, non-faulty secondaries will *always* be able to execute queries executed by the primary. If the primary's concurrency control isn't sufficiently blocking, a secondary might detect a conflict between queries that ran concurrently on the primary, and block one of them, potentially leading to a deadlock. CBS handles such situations by selectively aborting and later re-running transactions on the secondaries. After a sufficient number of aborts and re-runs, all transactions will be processed, thus ensuring that an insufficiently blocking primary does not compromise liveness, though it reduces system throughput.

3.4 View Changes

If the primary is faulty, CBS may be unable to make progress efficiently. The way to handle a faulty primary is to do a *view change* in which a new primary is selected by the coordinator and the old one is demoted to being a secondary. While any replica can be selected as the new primary, the best performance results from selecting one that is sufficiently blocking.

A correct view change must ensure that all transactions that committed prior to the view change continue to be committed afterwords. This condition is easy to satisfy: retain these transactions and abort all the others. A less draconian approach is to choose as the new primary a sufficiently blocking secondary that has completed processing all the statements in its pool; this way the aborts can be avoided. That secondary has agreed with all decisions made by the primary so far and therefore we can treat those decisions as if it made them. However, because the primary could be Byzantine faulty, there may be no secondary that is able to execute all statements in its pool. In this case, the coordinator times out waiting for a secondary to become eligible. Then it aborts all currently executing uncommitted transactions, and promotes a secondary that has committed all previous transactions.

It is always safe to do a view change, but view changes are undesirable since they can cause transactions to abort and interfere with the system making progress. The coordinator tries to decide when to do view changes intelligently but it doesn't always know when the primary is faulty. Clearly it knows the primary is faulty if it must abort a transaction due to incorrect results sent to a client. But if the primary doesn't respond, it might not be faulty; instead there might just be a network problem. Additionally, it may not be obvious which replica is faulty; for example, if the primary allows two statements to run in parallel, and a secondary blocks on one of them, this could mean that the secondary is faulty, or it could mean that the primary is faulty! Instead, the coordinator relies on heuristics: timers to limit waiting for the primary to respond, or counts of the number of times secondaries are blocking.

These heuristics need to ensure forward progress by guaranteeing the system will eventually choose a good replica as primary and will give it sufficient time to do its work. We use techniques similar to those that control view changes in BFT [6] for this task, *e.g.*, by choosing a new primary from all possible primaries and, when a series of view changes happen one after another, increasing timeouts exponentially between changes.

3.5 Fault Recovery

We now discuss how our system deals with failed nodes, *i.e.*, how it brings them back into a state where they can correctly participate in the protocol. We begin by discussing recovery of replicas that have suffered a crash failure. Then we discuss how to recover Byzantine-faulty replicas. Finally, we discuss recovery of the shepherd.

3.5.1 Recovery of a Crashed Replica

When a replica goes offline and then recovers, it rejoins the system in a consistent, but stale, state: all transactions that it did not commit prior to its failure have aborted locally. To become up-to-date, the replica must re-run any transactions that it aborted, but that were committed or are in progress in the system. All statements needed to recover the replica are in the pool of its manager.

A replica manager considers a replica to have crashed when the database connection to the replica breaks. If the manager had sent a COMMIT to the replica prior to learning that the connection had broken, but did not receive a reply, it cannot know whether that COMMIT had been processed before the replica went down. It cannot re-run the transaction if it already completed.

To allow the replica manager to determine what happened, we add a transaction log table to each replica; the table includes a row for each committed transaction, containing the commit barrier for that transaction. We also add a query to the end of each transaction to insert such a row in the table. If the transaction committed prior to the failure, the transaction log table will contain an entry for it. When the connection is re-established, the replica manager reads the transaction log table, compares the list of committed transactions with information maintained on the shepherd, and replays any missing committed transactions. After all the committed transactions have been processed, the manager starts running statements of in-progress transactions.

Adding this extra statement to each transaction will not result in deadlocks because each transaction only touches the log table on its last statement, after the coordinator has cleared it to commit. Also, the table need not be large. We can truncate a replica's log table periodically by determining the barrier, $T.b$, of the oldest transaction, T, not committed at this replica, and durably storing this information on the shepherd. Then we can overwrite the log table on that replica to contain only entries from $T.b$ on.

Of course, the database connection between the replica manager and a replica can break even when the replica has not failed. When such a network failure occurs, both parties will detect the failure (and each will assume that the other has failed). The replica will simply abort all pending transactions. When the shepherd re-initiates the connection to the replica after the failure heals, it re-runs transactions that have not yet committed, using information in the log table to avoid running transactions more than once. Hence, there is essentially no difference between how the shepherd handles a failure of a replica and how it handles a failure of the network between it and the replica.

Note finally that we can be smart about how we run transactions at slow replicas, including those that recover from crashes: we can avoid sending read-only transactions and read-only queries. By executing only state-changing queries, we can bring slow replicas up to date more quickly.

3.5.2 Recovery of a Byzantine-faulty Replica

Some mechanism is needed to detect and repair a Byzantine-faulty replica. This task must be done in a timely way, because otherwise our assumption of no more than f faulty replicas could be violated. Although a complete treatment of replica recovery is infeasible in this paper, we describe the basic solution we are developing here.

As discussed in Section 3.4, the coordinator cannot always know when a replica has suffered a Byzantine failure. Therefore we cannot rely on knowing about the failure in a timely way. For example, some faults are *silent*: the database state is corrupted, but the problem doesn't show up in the responses sent to the coordinator. Silent faults can occur due to hardware problems; they also occur during updates and even during the processing of supposedly read-only queries. Eventually these faults will be detected when the affected tables are read, but this might be so far in the future that it could too late (*i.e.*, by then more than f nodes might be corrupted). If the fault is due to an update, we could catch it by injecting a query to read what the update wrote, but this approach would significantly increase the cost of updates and would not catch other kinds of silent faults.

An additional problem is that the recovery mechanisms described in the previous sections aren't sufficient: they assume the replica is non-faulty, but out of date. If the replica is Byzantine, however, its state may be corrupted and rerunning transactions won't fix this problem. Instead we need a way to restore the state of the replica; after that, we can use the techniques in the previous section to bring it up to date.

Therefore we are exploring a *proactive* approach where a background scanning process compares the information stored in the different databases and then repairs databases found to be faulty. It can do this by using queries to identify faults in replicas. These queries are based on knowledge of what is stored in the databases: we simply compare the content of the tables. This "compare and repair" scheme exhaustively scans the replicas, identifies deviations in them, and executes repairs. In each compare and repair step, we rely on the assumption that at least $f + 1$ replicas are nonfaulty; this way we can recognize a faulty replica (since its state disagrees with that of the majority) and we can repair it (by replacing the bad state with what is stored at the majority).

For large databases, this recovery scheme could take a lot of time. We are exploring schemes to reduce this cost. The idea is to compare tables by comparing hashes of groups of their rows; preliminary results indicate that this scheme will perform quite well.

Each compare and repair step needs to run as a transaction, but there can be relatively few of these compared to transactions coming from clients; *i.e.*, we check in the background continuously but slowly. As long as these queries are run infrequently, or at idle times, their performance impact will be small.

3.5.3 Recovery from a Shepherd Crash

To survive crashes the shepherd maintains a write-ahead log. When the coordinator determines that it can commit a transaction, it writes the transaction queries (and their barriers), along with the COMMIT to the log. The coordinator forces the log to disk before replying to the client. The log is also forced to disk after logging a replica's transaction log table information prior to truncating the table. Note that the coordinator can employ the "group commit" optimization, where by delaying the processing of a COMMIT, the system can write a series of log records in one disk operation.

To recover from a crash, the shepherd reads the log, identifies all committed transactions, and initializes the statement pools at the managers to contain the statements of these

transactions. It knows which transactions to include in a replica manager's pool by examining the replica's transaction log table, just as if the replica had crashed. The shepherd can start accepting new client transactions when the replica selected as primary has completed executing all the statements in its pool.

The shepherd's log can be truncated by removing information about transactions that have committed at all the replicas. We can expect that most of the time all replicas are running properly and therefore the log need not be very large.

3.6 Correctness

In this section we present an informal discussion of the safety and liveness of our system. We assume here that no more than f replicas are faulty simultaneously.

3.6.1 Safety

CBS is safe because it guarantees that correct replicas have equivalent logical state, and that clients always get correct answers to transactions that commit.

Safety depends on the following assumption about databases: if correct replicas start in the same state and execute the same set of transactions in equivalent serial orders, they will end up in equivalent states and will produce identical answers to all queries in those transactions. This condition is satisfied by databases that ensure serializability. Safety additionally depends on our assumption that the replicas use two-phase locking.

The job of our system is to ensure that it presents transactions requested by clients to the databases in a way that ensures they will produce equivalent serial orders. The query-ordering rule ensures that each transaction runs at each replica as specified by the client; thus the transactions seen by the replicas are identical. Furthermore the commit-ordering rule ensures that COMMITs are sent to the replicas in a way that causes them to select equivalent serial orders. Suppose the commit-ordering rule allowed a transaction to commit only after all previous transactions have committed. All replicas will then implement identical commit orders, and since commit order is equivalent to serial order, all replicas thus implement equivalent serial orders. When the commit-ordering rule is relaxed to require only that all previous transactions are executed (but not committed), strict two-phase locking ensures that no two conflicting transactions will reach this point simultaneously. Thus we can be sure that correct replicas will have equivalent states after they execute the same set of transactions.

Additionally, clients receive only correct results for queries of transactions that commit because we vote on the answers: $f+1$ replicas must agree to each query response. The vote ensures that at least one correct replica agrees with the result, which implies that the result is correct. However, it is crucial that we delay the vote until each contributing replica is ready to commit the transaction, since only at that point can we be sure it is producing the answer that happens by executing that query at that place in the serial order. Strict two-phase locking ensures that, barring faults, a transaction that is ready to commit can be successfully committed with the answers that it has produced.

We continue to provide correct behavior even in the face of the various recovery techniques (recovery of a crashed replica, recovery from a database disconnection, recovery of the shepherd). The shepherd records complete information about committed transactions, and allowing it to recover from replica crashes and network disconnections; the log tables at the replicas are crucial to this recovery since they prevent transactions from executing multiple times. The write-ahead log at the shepherd ensures that the information about committed transactions isn't lost if the shepherd crashes.

View changes do not interfere with correct behavior because we retain information about committed transactions. Replica repair is also not an issue, since it replaces bad tables at a faulty replica with good information agreed to by $f+1$ replicas (and therefore agreed to by at least one non-faulty replica).

Finally note that the transaction-ordering rule is not needed for correctness. Instead, this rule is important for performance, because it ensures (assuming the primary is non-faulty and sufficiently blocking) that the secondaries execute queries in an order that avoids spurious deadlocks.

3.6.2 Liveness

Given a non-faulty primary, CBS is live assuming that messages are delivered eventually and correct replicas eventually process all messages they receive.

However, the primary may be faulty. We handle this case through the view change mechanism, which allows us to switch from a faulty primary to a non-faulty primary.

The problem with view changes is that there are cases where we can't be sure that the primary is faulty, and yet we do the view change anyway, *e.g.*, when the primary is merely slow to respond or secondaries are incorrectly ordering transactions. An adversary could cause us to do a view change in which a non-faulty primary is replaced by a faulty replica. However, the adversary cannot cause us to continuously do view changes, without making progress in between, because we exponentially increase the timeouts that govern when the next view change happens, and we select primaries in a way that will eventually lead to a non-faulty one. Eventually these timeouts will become large enough so that a non-faulty primary will be able to make progress, assuming that network delays cannot increase forever. Under this assumption (used for BFT [6]), our system is live.

4. HETEROGENEITY ISSUES

In this section we discuss two important issues that arise when heterogeneous database replicas are used. Recall that we require that each query presented by a client be processed identically by each replica. The first issue is language compatibility—different replicas may actually use different query languages. For example in HRDB our replicas all handle SQL queries. However, because database vendors implement vendor-specific SQL extensions, SQL is unfortunately not a "universal" standard and some translation may be required.

The second issue occurs even in a homogeneous deployment, though it is exacerbated by heterogeneity: some queries may execute non-deterministically. For example, because relations are unordered sets, different replicas may end up returning differently ordered responses to the same SQL query. This section discusses how HRDB addresses these issues in the context of SQL databases.

4.1 SQL Compatibility

No two databases implement identical variants of the SQL language. Although most of these differences are not in the implementation of the SQL ANSI standard, requiring applications to use only the standard core is restrictive. To address this SQL compatibility issue, we present two possibilities: an online solution and an offline alternative.

The online solution is for the shepherd to translate from client issued SQL into database-native SQL for each database replica to execute. There are existing software packages [22, 23] that accomplish this complex task. To improve performance, the shepherd could also maintain a cache of translated statements to avoid re-translating common statements. A database administrator could then look over the repository of cached translations and optimize them for better performance.

The offline solution is to hide non-standard SQL from the shepherd by burying it in views and stored procedures. Both views and stored procedures are often written by database administrators offline to improve performance. By independently implementing the same view or stored procedure on each heterogeneous database, the implementation details of the operation are removed from the SQL sent by the client application to HRDB. Of course, using views or stored procedures requires users of HRDB to implement their queries in several different database systems.

In general, either of these solutions may require some effort by users to port their SQL queries to several different database systems. Some effort may also be required to set up identical tables on different systems and optimize performance via creation of indices and other optimizations. We believe that these overheads are not overly onerous, especially given the increased robustness and fault tolerance that HRDB offers.

For the purposes of our implementation, we have built a basic translation layer for each database system that rewrites SQL queries using simple syntactic transformations (*e.g.*, translating date and time functions, handling various data definition language expressions, etc.).

4.2 Non-determinism

The coordinator needs to compare replies to SQL queries from database replicas to determine the correct answer. The coordinator compares both the result sets and the metadata returned for each query. However, SQL allows nondeterminism in how a database answers a query: rows of a result set may be returned in any order unless an order has been specified. In addition, some functions (like timestamps and auto-generated row IDs) return database-specific answers.

We solve the problem of non-deterministic row ordering by rewriting all queries to include an appropriate `ORDER BY` clause. This approach is likely to perform better than having the coordinator sort result sets because it can take advantage of the optimized sorting routines in the replica databases. Of course, the performance of some queries may degrade, and other solutions—hashing result sets and comparing the hashes, for example—might also be feasible.

Non-deterministic functions (like time stamp creation via calls to built-in functions executed at insert time) can result legitimate deviations in database state. In the case of timestamps, a fully featured coordinator could compute the value and rewrite the query itself. For our prototype, the application developer is responsible for writing queries that do not contain non-deterministic functions.

5. IMPLEMENTATION

We implemented the HRDB prototype in Java and it comprises about 8,000 total lines of code. Clients interact with the shepherd using JDBC, thus allowing HRDB to work with existing applications. The replica managers also use JDBC to interact with their replica databases. Configuring the shepherd only requires providing it the JDBC URLs of the databases to connect to. We have successfully run HRDB with MySQL (4.1 and 5.1), IBM DB2 V9.1, Microsoft SQLServer 2005 Express, and Derby 10.1. Two other common options, Oracle and PostgreSQL, use snapshot isolation and are thus incompatible.

JDBC access involves blocking operations: its calls do not return until the database returns an answer to the query. Thus, the shepherd must run each concurrent transaction in a separate thread on the replica manager. Overall, to support c concurrent transactions, the shepherd must use $c(2f+1)$ threads. Our implementation uses a thread pool to limit the number of threads and allocates threads to client connections using an asynchronous I/O mechanism to detect which clients have transactions ready for processing.

Using JDBC for the database replica interface also means that result sets must be unmarshaled in the replica manager and remarshaled to be sent to the client, since the wire format of each result set is database-specific. This step is also necessary to store the result set from the primary for comparison with the result sets from the secondaries. For large result sets, the unmarshal / remarshal step can incur a significant overheard.

We note that it might be too expensive to have each replica return the entire query result when the result is large. A way to reduce this cost is to have secondaries return a hash of the result instead. For example, the coordinator could rewrite the query for secondary replicas to nest the query statement in an aggregate that computes a hash over the entire result set. Most databases implement a standard hash function, like MD5, which could be used for this purpose. This optimization would significantly reduce the bandwidth requirements at the cost of increased computation at the database replicas.

6. ANALYSIS OF BUGS

This section first looks at the presence and character of bugs in databases, then describes HRDB's ability to tolerate and even discover bugs.

6.1 Bugs in Databases

We performed a simple analysis of bug reports in three database systems: DB2, Oracle and MySQL. Our goal was to understand to what extent reported bugs return incorrect answers to clients rather than cause crashes. Because it is hard to get access to authoritative information about bugs in practice, we looked at information available on vendor web sites about bug fixes. In all cases, we found thousands of bug reports, a rate consistent with other large software systems.

We divided the bugs into two broad categories: those that caused wrong results to be reported or the database to be corrupted and those that caused a crash. It is possible that crash-inducing bugs also caused wrong results to be re-

Bug category	DB2 2/03-8/06	Oracle 7/06-11/06	MySQL 8/06-11/06
DBMS crash	**120**	**21**	**60**
Non-crash faults	**131**	**28**	**64**
Incorrect answers	81	24	63
DB corruption	40	4	(inc. above)
Unauth. access	10	unknown	1

Table 1: Summary of bugs reported in different systems. In all cases, over 50% of the reported bugs cause non-crash faults resulting in incorrect answers to be returned to the client, database corruption, or unauthorized accesses. Current techniques only handle crash faults. The numbers for the different systems are reports over different time durations, so it is meaningless to compare them across systems.

ported; we count these as crashes. Table 1 summarizes the results.

DB2: We looked at the set of bugs (called "Authorized Program Analysis Reports", or APARs) reported as being fixed in DB2 UDB 8 Fixpaks 1 through 13; these were released between February 2003 and August 2006. There are several thousand such reports—many of the reports are related to tools associated with DB2, or with the optimizer generating inefficient query plans. However, a number of them result in database corruption or incorrect answers that neither the user nor the database system itself would detect. Examples of such bugs include reports titled "incorrect results returned when a query joins on char columns of different lengths and uses like predicate with string constant", "query compiler incorrectly removes a correlated join predicate leading to an incorrect result set", and "in some rare cases, reorg operations may result in corrupted index".

Of the 251 bugs reported, the majority (131) caused non-crash faults.

Oracle: We studied bugs that were fixed between July and November 2006 for an upcoming release (labeled version 11.1 at the time) of the Oracle Server Enterprise Edition by searching the Oracle Bug Database on the Oracle Metalink Server. We did not characterize security bugs, as these are reported separately from the database engine. The number of our sampled bugs that cause wrong results or database corruption (28) exceeds the number that cause crashes (21).

MySQL: We analyzed 90 days of bug reports from MySQL between August 15 and November 13, 2006. Of the 900 issues reported, about 131 of them are verified issues inside the database server. As with DB2 and Oracle, more than half of the verified MySQL database system engine bugs result in wrong answers.

Because current recovery and replication techniques only handle bugs that immediately cause crashes, they apply to less than half of the bugs seen in this informal study. We also note that our results are consistent with those reported by Gashi *et al.* in a recent and more detailed study [10] of 181 bugs in PostgreSQL 7.0 and 7.2, Oracle 8.0.5, and SQL Server 7. They tested these systems and found that 80 of the reported bugs result in "non-self-evident" faults—*i.e.*, in incorrect answers without crashes or error messages. They also report that for all but five bugs, the bug occurred in only one of the four systems they tested, and in no case did any bug manifest itself in more than two systems.

6.2 Tolerance of Bugs with HRDB

To demonstrate HRDB's ability to tolerate non-fail-stop failures, we attempted to reproduce some of the bugs found in our study. We focused on bugs that could be triggered by specific SQL statements as they are easier to exhibit, reserving a more general study for future work. We note that we only attempted to reproduce a small number of bugs.

We successfully reproduced seven such bugs in the production versions of database systems that we used (see Figure 6). We ran our system with the bugs using the configurations shown in the figure and we were able to successfully mask the failure in each case. Some bugs used SQL that was supported by all vendors; we were able to mask these bugs using databases from different vendors, since none of them appeared across vendors. Other bugs involved vendor-specific syntax and thus were applicable only to certain databases. We were able to mask bugs in MySQL 4.1.16 using MySQL 5.1.11, and to mask bugs in MySQL 5.1.11 using MySQL 4.1.16, demonstrating the utility of our system even within different versions of the same database.

6.3 Discovery of Bugs using HRDB

While producing the performance results detailed in following section, we noticed that when running HRDB with 3 identical MySQL replicas, the primary was producing answers that did not match the secondaries' answers. Upon further investigation, we found that MySQL was allowing phantom reads: a query in transaction T1 was seeing inserts from transaction T2 despite T2 being ordered after T1. Isolation level SERIALIZABLE should not permit phantom reads.

We submitted the details of the case to the MySQL development team, and they quickly verified that it was a bug. MySQL failed to acquire the correct locks in `SELECT` queries that use an `ORDER BY ... DESC`. The bug affected every version of MySQL since 3.23 and they quickly proposed a patch.

The discovery of this bug demonstrates HRDB's ability to find bugs. In this case, the bug was non-deterministic (timing dependent), and from looking at the MySQL bug database, possibly noticed previously but dismissed as unreproducible.

7. PERFORMANCE ANALYSIS

In this section, we evaluate the performance of HRDB under a high-concurrency workload. Our tests were done with $2f + 1 = 3$ replicas, 1 primary and two secondaries, which can tolerate 1 faulty replica. Our implementation does not perform logging on the shepherd. The tests were run on a cluster of Dell Poweredge 1425 machines running Fedora Core 4 Linux, each with dual processor 2.8 GHz Xeons with 2GB of RAM and SATA 160 GByte disks, and all attached to the same gigabit Ethernet switch. Some tests also included a Dell OPTIPLEX GX620 3.8Ghz with 4GB of RAM running Windows XP. The databases used were MySQL 4.1.22-max and 5.1.16 with InnoDB tables, IBM DB2 V9.1, and Commercial Database X[2]. We also ran experiments with Apache Derby 10.1.3.1 (a Java-based database) but omit performance numbers here as it is much slower than the other systems.

[2]Database name omitted due to licensing constraints

Bug	Description	Faults	Doesn't Fault
MySQL #21904	*Parser problem with IN() subqueries*	MySQL 4.1, 5.1	DB2, Derby
MySQL #7320	*Aggregates not usable where they should be*	MySQL 4.1	MySQL 5.1
MySQL #13033	*INNER JOIN and nested RIGHT OUTER JOIN*	MySQL 4.1	MySQL 5.1
MySQL #24342	*MERGE tables use wrong value when checking keys*	MySQL 5.1	MySQL 4.1
MySQL #19667	*GROUP BY containing cast to DECIMAL*	MySQL 5.1	MySQL 4.1
DB2 #JR22690	*Query optimizer optimizes away things it shouldn't*	DB2 V9.0	MySQL 4.1
Derby #1574	*UPDATE with COALESCE and subquery*	Derby 10.1.3	MySQL 5.1, DB2

Figure 6: Bugs we reproduced and masked with HRDB.

We use the TPC-C [25] query mix to test our system because it produces a high concurrency transaction processing workload with non-trivial transaction interaction. TPC-C simulates an order-entry environment where a set of clients issue transactions against a database. The TPC-C workload is heavily slanted towards read/write transactions, with read-only transactions comprising only 8% of the workload. Our implementation uses the same transaction types but does not attempt to model the keying, wait, and think times. Instead, each client waits for a set period of time (between 180 ms and 220 ms).

Our main objective with these experiments is to measure the overhead of the HRDB shepherd and to determine how much concurrency the commit barrier scheduling protocol is able to preserve in a typical online transaction processing workload. We note that the HRDB implementation is an untuned Java prototype and the TPC-C implementation is also untuned.

7.1 HRDB Overhead

To measure the overhead associated with the HRDB shepherd, we compared running the TPC-C workload against a database directly to running it through the HRDB shepherd. The database is loaded with 30 warehouses and 10 districts per warehouse to provide ample concurrency. We used small (1/6th size) warehouses so our untuned TPC-C implementation would achieve reasonable throughput on our hardware. The results are shown in Figure 7. The error bars show the standard deviation of 7 runs for each data point.

To determine the sources of performance overhead, we ran the shepherd in a number of different modes. In *Passthrough* mode, the shepherd has only one database replica, running MySQL. Since it has no secondaries it does not use any replication machinery or commit barrier scheduling; hence it represents the overhead of using a shepherd. *Shepherd 3DBs, CBS* represents the performance of HRDB using commit barrier scheduling with three MySQL 5.1 database replicas. *Shepherd 3DBs, SS* (Sequential Secondaries) is an execution mode modeled after the Pronto [19] system, where the coordinator waits until the commit is received before executing the transaction on the secondaries. Furthermore, the secondaries execute transactions sequentially, thus avoiding any possibility of incorrectly ordering transactions. In *Shepherd 3DBs, Serial* mode, transactions are executed one at a time on the entire replica set.

Unsurprisingly, the best performance is achieved by interacting with the database directly. The *Passthrough* line shows that there is negligible overhead from the middleware. *CBS* performs very well (about 17% performance loss at saturation), demonstrating the effectiveness of CBS at extracting concurrency. Most of the performance loss comes from waiting for agreement before committing the transaction. Since the transaction holds locks on the primary during the waiting period, other conflicting transactions are prevented from making forward progress. Thus additional latency prior to commit reduces the available concurrency in the workload, reducing throughput.

The *SS* performance (52% overhead) demonstrates that the additional complexity of CBS provides substantial performance benefits. *SS* does not need a sufficiently blocking primary, nor must it ever abort and retry transactions on the secondaries due to ordering conflicts. However, *SS* cannot perform transaction execution pipelining, forcing it to wait longer for agreement before commit. Additionally, it lacks concurrent execution at the secondaries. These two issues result in *SS* running 42% slower than *CBS* at saturation.

Finally, the *Serial* performance (91% overhead) is limited because it does not scale as more clients are added. Note that a naive SQL analysis based on table-locking to determine which transactions do not conflict would have similar performance to *Serial* on this workload. The two most common transaction types in TPC-C (accounting for 88% of the workload) both write to the same table.

We also experimented with varying the wait time and number of warehouses. Changes in wait time shift the number of clients required to reach saturation, but do not affect the relative performance of CBS with respect to MySQL. Reducing the number of warehouses increases transaction contention for locks. Under high contention, transaction throughput is inversely proportional to transaction latency. Since the shepherd (and *CBS*) introduce additional latency, the shepherd overhead increases significantly in high contention situations.

7.2 Heterogeneous Replication

Now we demonstrate that HRDB is capable of using a heterogeneous replica set, and that a sufficiently blocking primary is a reasonable, but not necessary, assumption. We implemented the TPC-C workload in standard SQL, and thus were able to run it against MySQL 4.1 and 5.1, DB2, and Commercial Database X. Our implementation does simple rewriting of SQL queries to allow the benchmark to work, including handling issues with NULL ordering, column types, and different function names. We ran the test with 20 clients on a 10 (small) warehouse workload, with the objective of demonstrating feasibility and relative performance of various configurations (not databases).

Figure 8 shows the throughput results for a number of configurations. The *Direct* configuration is one where the test harness interacts directly with the database. The *Passthrough* configuration adds an intermediary that performs query and result forwarding. *Primary* is a 3 database

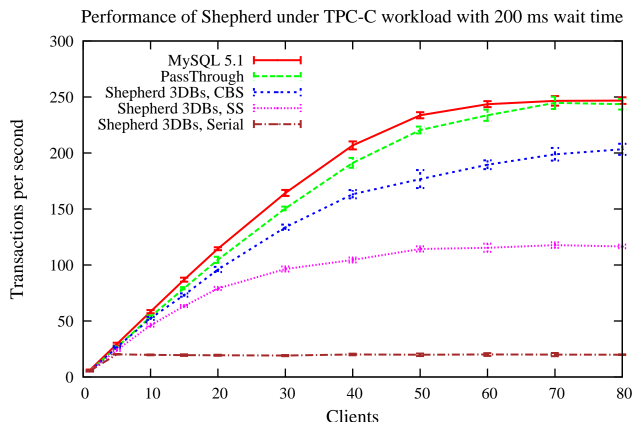

Figure 7: Performance on TPC-C workload as a function of number of clients, using 200 ms wait time between transaction submits (per client), on a homogeneous replica set.

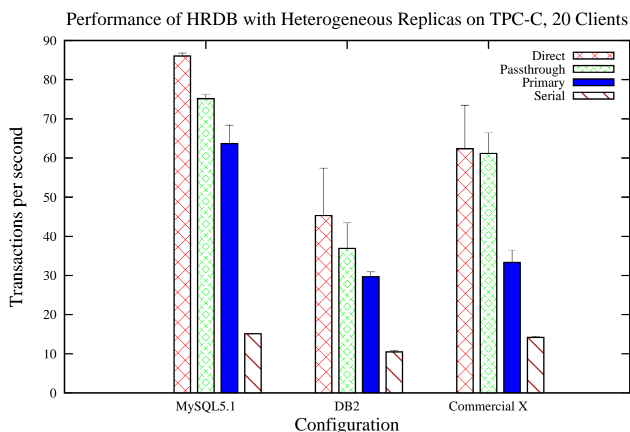

Figure 8: Performance of HRDB with a heterogeneous replica set (MySQL, DB2, Commercial Database X).

(MySQL 5.1, DB2, Commercial Database X) configuration with the given database as the primary. *Serial* is a 3 database (MySQL 5.1, DB2, Commercial Database X) configuration, where transactions are executed serially (without concurrency).

In the absence of faults and with a sufficiently blocking primary, HRDB runs at the speed of the slowest of the $f+1$ fastest replicas or the primary, whichever is slower. With MySQL as the primary, the slowest of the $f+1$ fastest is Commercial Database X, and the system performs as expected. With DB2 as the primary, the primary is the bottleneck. Commercial Database X is not *sufficiently blocking* for MySQL; if it is the primary, this causes MySQL to block on one of the statements of the workload. With MySQL blocked, the shepherd must wait for DB2 and thus the performance is capped by DB2's performance.

7.3 Fail-stop faults

To demonstrate that HRDB survives fail-stop faulty replicas, we measured transaction throughput while one replica

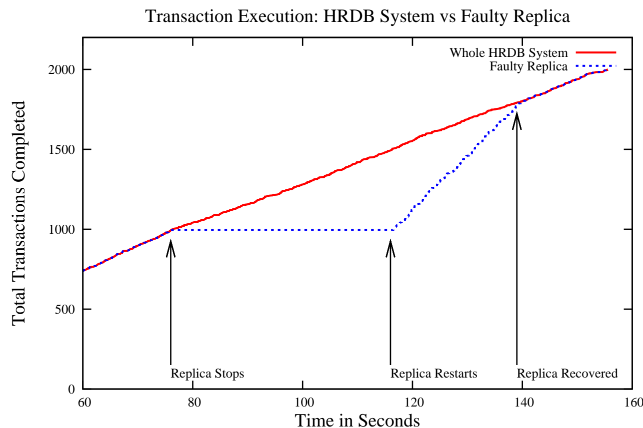

Figure 9: Transactions completed over time on TPC-C workload as one replica is crashed and restarted

was crashed and restarted. We ran a 20 client TPC-C workload and recorded the timestamps at which each replica committed each transaction. Figure 9 shows a replica being crashed 76 seconds into the test and restarted 40 seconds later. In this time, the rest of the system continues executing because it can tolerate a faulty replica. When the faulty replica restarts, it is 500 transactions behind.

For the overall system, the performance impact of a crashed replica is negligible. When the replica restarts, it is able to catch up with the rest of the replicas because it does not execute any reads. For this workload, the slow replica was able to execute transactions at nearly three times the rate of the rest of the replicas (750 vs. 250 transactions in 20 seconds). This result demonstrates that transiently slow or rebooted replicas can catch up with the other replicas.

8. CONCLUSION

HRDB shows, for the first time, a practical way to tolerate Byzantine faults in transaction processing systems while ensuring that transaction statements can execute concurrently on the replicas. The key idea is commit barrier scheduling, which allows the coordinator in the shepherd to observe transaction execution on the primary and ensure that all non-faulty secondaries execute transactions in an equivalent serial order. CBS ensures that all replicas have the same logical state and that clients get correct answers for all committed transactions.

We have implemented CBS as part of HRDB, which provides a single-copy serializable view of a replicated relational database system using unmodified production versions of several commercial and open-source databases as the replicas. Our experiments show that HRDB can tolerate one faulty replica with only a modest performance overhead (about 17% for the TPC-C benchmark on a homogeneous replica set). Furthermore, HRDB is 10 times faster on TPC-C than a naive implementation that forcibly serializes all transactions. We also showed how HRDB is able to mask faults observed in practice, including one that we uncovered in MySQL.

HRDB has a number of attractive properties. First, it requires no modifications to the database systems themselves. Second, it does not require any analysis of SQL queries to detect conflicts, which is intractable for complex queries at

finer granularities than table-level conflicts, and which may not work at all for materialized views or stored procedures. Third, HRDB is able to preserve substantial amounts of concurrency in highly concurrent database workloads.

We conclude this paper by outlining some avenues for future work. The most immediate future work items for us are to: (1) develop, implement, and evaluate efficient algorithms for reconciling divergent database replicas, (2) explore scheduling algorithms for alternate concurrency control schemes (*e.g.*, snapshot isolation), (3) further reduce shepherd overhead by implementing programmatic and algorithmic optimizations, and (4) develop methods for Byzantine replication of the shepherd while preserving reasonable performance.

9. ACKNOWLEDGMENTS

We would like to thank our paper shepherd Mike Dahlin, and the anonymous reviewers for their insightful comments.

This research was supported NSF ITR grants CNS-0428107 and CNS-0225660, and by T-Party, a joint program between MIT and Quanta Computer Inc., Taiwan. Ben Vandiver was awarded an SOSP student travel scholarship, supported by Infosys, to present this paper at the conference.

10. REFERENCES

[1] A. E. Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *TODS*, 14(2):264–290, 1989.

[2] C. Amza, A. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. International Middleware Conference. ACM/IFIP/Usenix, June 2003.

[3] J. F. Bartlett. A NonStop kernel. In *SOSP '81: Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, New York, NY, USA, 1981. ACM Press.

[4] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *TODS*, 9(4):596–615, 1984.

[5] A. Bhide, A. Goyal, H.-I. Hsiao, and A. Jhingran. An efficient scheme for providing high availability. In *SIGMOD*, pages 236–245, 1992.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[7] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3), Aug. 2003.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Conference*, 2004.

[9] H. Garcia-Molina and B. Kogan. Achieving high availability in distributed databases. *IEEE Trans. Softw. Eng.*, 14(7):886–896, 1988.

[10] I. Gashi, P. Popov, and L. S. V. Stankovic. On designing dependable services with diverse off-the-shelf SQL servers. *Lecture Notes in Computer Science*, 3069:191–214, 2004.

[11] Goldengate. `http://www.goldengate.com/technology/architecture.html`.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, pages 173–182, 1996.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1992.

[14] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *TODS*, 25(3):333–379, 2000.

[15] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks*, 2004.

[16] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–430, New York, NY, USA, 2005. ACM Press.

[17] H. G. Molina, F. Pittelli, and S. Davidson. Applications of Byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, 1986.

[18] M. Patino-Martinez, R. Jiminez-Peris, B. Kemme, and G. Alonso. Middle-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[19] F. Pedone and S. Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. Technical Report HPL-2000-96, HP Laboratories Palo Alto, 2000.

[20] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: replicated database services for world-wide applications. In *SIGOPS European workshop*, pages 275–280, 1996.

[21] C. Plattner and G. Alonso. Ganymede: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[22] SQLFairy. `http://sqlfairy.sourceforge.net/`.

[23] Swissql—SQLOne Console 3.0. `http://www.swissql.com/products/sql-translator/sql-converter.html`.

[24] Sybase replication server. `http://www.sybase.com/products/businesscontinuity/replicationserver/`.

[25] Transaction Processing Performance Council. TPC-C. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`.

[26] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, New York, NY, USA, 2003. ACM Press.