Byzantium: Byzantine-Fault-Tolerant Database Replication Providing Snapshot Isolation*

Nuno Preguiça[1] Rodrigo Rodrigues[2] Cristóvão Honorato[3] João Lourenço[1]
[1] CITI/DI-FCT-Univ. Nova de Lisboa
[2] Max Planck Institute for Software Systems (MPI-SWS)
[3] INESC-ID and Instituto Superior Técnico

Abstract

Database systems are a key component behind many of today's computer systems. As a consequence, it is crucial that database systems provide correct and continuous service despite unpredictable circumstances, such as software bugs or attacks. This paper presents the design of Byzantium, a Byzantine fault-tolerant database replication middleware that provides snapshot isolation (SI) semantics. SI is very popular because it allows increased concurrency when compared to serializability, while providing similar behavior for typical workloads. Thus, Byzantium improves on existing proposals by allowing increased concurrency and not relying on any centralized component. Our middleware can be used with off-the-shelf database systems and it is built on top of an existing BFT library.

1 Introduction

Transaction processing database systems form a key component of the infrastructure behind many of today's computer systems, such as e-commerce websites or corporate information systems. As a consequence, it is crucial that database systems provide correct and continuous service despite unpredictable circumstances, which may include hardware and software faults, or even attacks against the database system.

Applications can increase their resilience against faults and attacks through Byzantine-fault-tolerant (BFT) replication. A service that uses BFT can tolerate arbitrary failures from a subset of its replicas. This not only encompasses nodes that have been attacked and became malicious, but also hardware errors, or software bugs. In particular, a recent study [13] showed that the majority of bugs reported in the bug logs of three commercial database management systems would cause the system to fail in a non-crash manner (i.e., by providing incorrect answers, instead of failing silently). This supports the claim that BFT replication might be a more adequate technique for replicating databases, when compared to traditional replication techniques that assume replicas fail by crashing [2].

In this paper we propose the design of Byzantium, a Byzantine-fault-tolerant database replication middleware. Byzantium improves on existing BFT replication for databases both because it has no centralized components (of whose correctness the integrity of the system depends) and by allowing increased concurrency, which is essential to achieve good performance.

The main insight behind our approach is to aim for weaker semantics than traditional BFT replication approaches. While previous BFT database systems tried to achieve strong semantics (such as linearizability or 1-copy serializability [2]), Byzantium only ensures snapshot isolation (SI), which is a weaker form of semantics that is supported by most commercial databases (e.g., Oracle, Microsoft SQL Server). Our design minimizes the number of operations that need to execute the three-phase agreement protocol that BFT replication uses to totally order requests, and allows concurrent transactions to execute speculatively in different replicas, to increase concurrency.

1.1 Related Work

The vast majority of proposals for database replication assume the crash failure model, where nodes fail by stopping or omitting some steps (e.g., [2]). Some of these works also focused on providing snapshot isolation to improve concurrency [11, 10, 5]. Assuming replicas fail by crashing simplifies the replication algorithms, but does not allow the replicated system to tolerate many of the faults caused by software bugs or malicious attacks.

There are few proposals for BFT database replication. The schemes proposed by Garcia-Molina et al. [7] and by Gashi et al. [8] do not allow transactions to execute concurrently, which inherently limits the performance of the system. We improve on these systems by showing how ensuring weaker semantics (snaphost isolation) and bypassing the BFT replication protocol whenever possible allows us to execute transactions concurrently.

HRDB [13] is a proposal for BFT replication of off-the-shelf databases which uses a trusted node to coor-

*This work was supported by FCT/MCTES, project # PTDC/EIA/74325/2006.

dinate the replicas. The coordinator chooses which requests to forward concurrently, in a way that maximizes the amount of parallelism between concurrent requests. HRDB provides good performance, but requires trust in the coordinator, which can be problematic if replication is being used to tolerate attacks. Furthermore, HRDB ensures 1-copy serializability, whereas our approach provides weaker (yet commonly used) semantics to achieve higher concurrency and good performance.

1.2 Paper Outline

The remainder of the paper is organized as follows. Section 2 presents an overview of the system. Section 3 describes its design. Section 4 discusses correctness. Section 5 addresses some implementation issues, and Section 6 concludes the paper.

2 Byzantium Overview

2.1 System model

Byzantium uses the PBFT state machine replication algorithm [3] as one of its components, so we inherit the system model and assumptions of this system. Thus, we assume a Byzantine failure model where faulty nodes (client or servers) may behave arbitrarily. We assume the adversary can coordinate faulty nodes but cannot break cryptographic techniques used. We assume at most f nodes are faulty out of $n = 3f + 1$ replicas.

Our system guarantees safety properties in any asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, corrupt them, delay them arbitrarily, or deliver them out of order. Liveness is only guaranteed during periods where the delay to deliver a message does not grow indefinitely.

2.2 Database model

In a database, the state is modified by applying transactions. A transaction is started by a BEGIN followed by a sequence of read or write operations, and ends with a COMMIT or ROLLBACK. When issuing a ROLLBACK, the transaction aborts and has no effect on the database. When issuing a COMMIT, if the commit succeeds, the effects of write operations are made permanent in the database.

Different semantics (or *isolation levels*) have been defined for database systems [1], allowing these systems to provide improved performance when full serializability is not a requirement. Byzantium provides the *snapshot isolation* (SI) level. In SI, a transaction logically executes in a database snapshot. A transaction can commit if it has no write-write conflict with any committed concurrent transaction. Otherwise, it must abort.

SI allows increased concurrency among transactions when compared with serializability. For example, when enforcing serializability, if a transaction writes some data item, any concurrent transaction that reads the same data item cannot execute (depending on whether the database uses a pessimistic or optimistic concurrency control mechanism, the second transaction will either block until the first one commits or will have to abort due to serializability problems at commit time). With SI, as only write-write conflicts must be avoided, both transactions can execute concurrently. This difference not only allows increased concurrency for transactions accessing the same data items, but it is also beneficial for read-only transactions, since they can always execute without ever needing to block or to abort.

The SI level is very popular, as many commercial database systems implement it and it has been shown that for many typical workloads (including the most widely used database benchmarks, TPC-A, TPC-B, TPC-C, and TPC-W), the execution under SI is equivalent to strict serializability [4]. Additionally, is has been shown how to transform a general application program so that its execution under SI is equivalent to strict serializability [6].

2.3 System Architecture

Byzantium is built as a middleware system that provides BFT replication for database systems. The system architecture, depicted in Figure 1, is composed by a set of $n = 3f + 1$ servers and a finite number of clients.

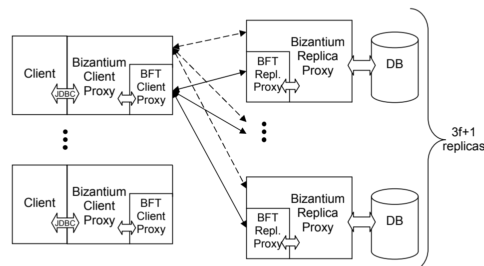

Figure 1: System Architecture.

Each server is composed by the Byzantium replica proxy, which is linked to the PBFT replica library [3], and a database system. The database system maintains a full copy of the database. The replica proxy is responsible for controlling the execution of operations in the database system. The replica proxy receives inputs from both the PBFT replication library (in particular, it provides the library with an `execute` upcall that is called after client requests run through the PBFT protocol and are ready to be executed at the replicas), and it also receives messages directly from the Byzantium clients (which are not serialized by the PBFT protocol).

The database system used in each server can be differ-

```
1  function db_begin() : trxHandle
2    uid = generate new uid
3    coord_replica = select random replica
4    opsAndHRes = new list
5    BFT_exec( <BEGIN,uid,coord_replica>)
6    trxHandle = new trxHandle( uid, coord_replica,
7                                          opsAndHRes)
8    return trxHandle
9  end function
10
11 function db_op( trxHandle, op) : result
12   result = replica_exec( trxHandle.coord_replica,
13                                 <trxHandle.uid,op>)
14   trxHandle.opsAndHRes.add( <op,H(result)>)
15   return result
16 end function
17
18 function db_commit( trxHandle)
19   result = BFT_exec( <COMMIT,trxHandle.uid,
20                              trxHandle.opsAndHRes>)
21   if( res == true)
22     return
23   else
24     throw ByzantineExecutionException
25   endif
26 end function
```

Figure 2: Byzantium client proxy code.

```
1  upcall FOR BFT_exec( <BEGIN,uid,coord_replica>)
2    DB_trx_handle = db.begin()
3    openTrxs.put( uid, <DB_trx_handle,coord_replica>)
4  end upcall
5
6  upcall for BFT_exec( <COMMIT,uid,cltOpsAndHRes>)
7                                          : boolean
8    <DB_trx_handle,coord_replica> = openTrxs.get( uid)
9    openTrxs.remove( uid)
10   if( coord_replica != THIS_REPLICA)
11     execOK = exec_and_verify( DB_trx_handle,
12                                     cltOpsAndHRes)
13     if( NOT execOK)
14       DB_trx_handle.rollback()
15       return false
16     endif
17   endif
18   if( verifySIProperties( DB_trx_handle))
19     DB_trx_handle.commit()
20     return true
21   else
22     DB_trx_handle.rollback()
23     return false
24   endif
25 end upcall
26
27 upcall for replica_exec( <uid,op>) : result
28   <DB_trx_handle,coord_replica> = openTrxs.get( uid)
29   return DB_trx_handle.exec( op)
30 end upcall
```

Figure 3: Byzantium replica proxy code.

ent, to ensure a lower degree of fault correlation, in particular if these faults are caused by software bugs [12, 13]. The only requirement is that they all must implement the *snapshot isolation* semantics and support savepoints[1], which is common in most database systems.

Users applications run in client nodes and access our system using a standard database interface (in this case, the JDBC interface). Thus, applications that access conventional database systems using a JDBC interface can use Byzantium with no modification. The JDBC driver we built is responsible for implementing the client side of the Byzantium protocol (and thus we refer to it as the Byzantium client proxy). Some parts of the client side protocol consist of invoking operations that run through the PBFT replication protocol, and therefore this proxy is linked with the client side of the PBFT replication library.

In our design, PBFT is used as a black box. This enables us to easily switch this replication library with a different one, provided both offer the same guarantees (i.e., state machine replication with linearizable semantics) and have a similar programming interface.

3 System Design

3.1 System operation

In this section, we describe the process of executing a transaction. We start by assuming that clients are not Byzantine and address this problem in the next section. The code executed by the client proxy is presented in Figure 2 and the code executed by the replica proxy is presented in Figure 3. We omitted some details (such as error and exception handling) from the code listing for simplicity.

The approach taken to maximize concurrency and improve performance is to restrict the use of the PBFT protocol to only the operations that need to be totally ordered among each other. Other operations can execute speculatively in a single replica (that may be faulty and provide incorrect replies) and we delay validating these replies until commit time.

The application program starts a transaction by executing a BEGIN operation (*function db_begin*, Figure 2, line 1). The client starts by generating a unique identifier for the transaction and selecting a replica responsible to speculatively execute the transaction – we call this the coordinator replica for the transaction or simply coordinator. Then, the client issues the corresponding BFT operation to execute in all replicas (by calling the *BFT_exec*($< BEGIN, ... >$) method from the PBFT library, which triggers the corresponding upcall at all replicas, depicted in Figure 3, line 1). At each replica, a database transaction is started. Given the properties of the PBFT system, and as both BEGIN and COMMIT operations execute serially as PBFT operations, this guarantees that the transaction is started in the same (equivalent) snapshot of the database in every correct replica.

After executing BEGIN, an application can execute a sequence of read and write operations (*function db_op*, Figure 2, line 11). Each of these operations executes only in the coordinator of the transaction (by calling

[1] A savepoint allows the programmer to declare a point in a transaction to which it can later rollback.

replica_exec, which triggers the corresponding upcall at the coordinator replica, depicted in Figure 3, line 27). The client proxy stores a list of the operations and corresponding results (or a secure hash of the result, if it is smaller).

The transaction is concluded by executing a COMMIT operation (*function db_commit*, Figure 2, line 18). The client simply issues the corresponding BFT operation that includes the list of operations of the transaction and their results. At each replica, the system verifies if the transaction execution is valid before committing it (by way of the *BFT_exec*($< COMMIT, ... >$) upcall, Figure 3, line 6).

To validate a transaction prior to commit, the following steps are executed. All replicas other than the primary have to execute the transaction operations and verify that the returned results match the results previously obtained in the coordinator. Given that the transaction executes in the same snapshot in every replica (as explained in the BEGIN operation), if the coordinator was correct, all other correct replicas should obtain the same results. If the coordinator was faulty, the results obtained by the replicas will not match those sent by the client. In this case, correct replicas will abort the transaction and the client throws an exception signaling Byzantine behavior. In Section 5, we discuss some database issues related with this step.

Additionally, all replicas including the coordinator, need to verify if the SI properties hold for the committing transaction. This verification is the same that is executed in non-byzantine database replication systems (e.g. [5]) and can be performed by comparing the write set of the committing transaction with the write sets of transactions that have previously committed after the beginning of the committing transaction. As this process is deterministic, every correct replica will consequently either commit or abort the transaction.

A transaction can also end with a ROLLBACK operation. A straightforward solution is to simply abort transaction execution in all replicas. We discuss the problems of this approach and propose an alternative in Section 3.4.

3.2 Tolerating Byzantine clients

The system needs to handle Byzantine clients that might try to cause the replicated system to deviate from the intended semantics. Note that we are not trying to prevent a malicious client from using the database interface to write incorrect data or delete entries from the database. Such attacks can be limited by enforcing security/access control policies and maintaining additional replicas that can be used for data recovery [9].

As we explained, PBFT is used by the client to execute operations that must be totally ordered among each other. Since PBFT already addresses the problem of Byzantine client behavior in each individual operation, our system only needs to address the validity of the operations that are issued to the database engines running in the replicas.

First, replicas need to check if they are receiving a valid sequence of operations from each client. Most checks are simple, such as verifying if a BEGIN is always followed by a COMMIT/ROLLBACK and if the unique identifiers that are sent are valid.

There is one additional aspect that could be exploited by a Byzantine client: the client first executes operations in the coordinator and later propagates the complete sequence of operations (and results) to all replicas. At this moment, the coordinator does not execute the operations, as it has already executed them. A Byzantine client could exploit this behavior by sending a sequence of operations during the COMMIT PBFT requests that is different from the sequence of operations that were previously issued to the coordinator, leading to divergent database states at the coordinator and the remaining replicas.

To address this problem, while avoiding a new round of message among replicas, we have decided to proceed with transaction commitment using the latest sequence of operations submitted by the client.

The code executed by the replica proxy for supporting Byzantine clients is presented in Figure 4. To be able to compare if the sequence of operations submitted initially is the same that is submitted at commit time, the coordinator also logs the operations and their results as they are executed (line 42). At commit time, if the received list differs from the log, the coordinator discards executed operations in the current transaction and executes operations in the received list, as any other replica.

For discarding the executed operations in the current transaction, we rely on a widely available database mechanism, *savepoints*, that enables rolling back all operations executed inside a running transaction after the savepoint is established. When the BEGIN operation executes, a savepoint is created in the initial database snapshot (line 3). Later, when it is necessary to discard executed operations but still use the same database snapshot, the transaction is rolled back to the savepoint previously created (line 17). This ensures that all replicas, including the coordinator, execute the same sequence of operations in the same database snapshot, guaranteeing a correct behavior of our system.

3.3 Tolerating a faulty coordinator

A faulty coordinator can return erroneous results or fail to return any results to the clients. The first situation is addressed by verifying, at commit time, the correctness of results returned to all replicas, as explained pre-

```
upcall FOR BFT_exec( <BEGIN, uid , coord_replica >)
  DB_trx_handle = db.begin()
  DB_trx_handle.setSavepoint( 'init')
  opsAndHRes = new list
  openTrxs.put( uid, <DB_trx_handle, coord_replica,
                                           opsAndHRes>)
end upcall

upcall for BFT_exec( <COMMIT, uid , cltOpsAndHRes>)
                                                  : boolean
  <DB_trx_handle, coord_replica, opsAndHRes> =
                                      openTrxs.get( uid)
  openTrxs.remove( uid)
  hasToExec = coord_replica != THIS_REPLICA
  if( coord_replica == THIS_REPLICA)
    if( different_list( cltOpsAndHRes, opsAndHRes))
      DB_trx_handle.rollbackToSavepoint( 'init')
      hasToExec = true
    endif
  endif
  if( hasToExec)
    execOK = exec_and_verify(DB_trx_handle,
                                          cltOpsAndHRes)
    if( NOT execOK)
      DB_trx_handle.rollback()
      return false
    endif
  endif
  if( verifySIProperties( DB_trx_handle))
    DB_trx_handle.commit()
    return true
  else
    DB_trx_handle.rollback()
    return false
  endif
end upcall

upcall for replica_exec( <uid ,op>) : result
  <DB_trx_handle, coord_replica, opsAndHRes> =
                                      openTrxs.get( uid)
  result = DB_trx_handle.exec( op)
  opsAndHRes.add(<op,H(res)>)
  return result
end upcall
```

Figure 4: Byzantium replica proxy code, supporting Byzantine clients.

viously. This guarantees that correct replicas will only commit transactions for which the coordinator has returned correct results for every operation.

If the coordinator fails to reply to an operation, the client selects a new coordinator to replace the previous one and starts by re-executing all previously executed operations of the transaction in the new coordinator. If the obtained results do not match, the client aborts the transaction by executing a ROLLBACK operation and throws an exception signaling Byzantine behavior. If the results match, the client proceeds by executing the new operation.

At commit time, a replica that believes to be the coordinator of a transaction still verifies that the sequence of operations sent by the client is the same that the replica has executed. Thus, if a coordinator that was replaced is active, it will find out that additional operations have been executed. As explained in the previous section, it will then discard operations executed in the current transaction and it will execute the list of received operations, as any other replica. This ensures a correct behavior of our system, as all replicas, including replaced coordinators, execute the same sequence of operations in the same database snapshot.

3.4 Handling aborted transactions

When a transaction ends with a ROLLBACK operation, a possible approach is to simply abort the transaction in all replicas without verifying if previously returned results were correct (e.g., this solution is adopted in [13]). In our system, this could be easily implemented by executing a BFT operation that aborts the transaction in each replica.

This approach does not lead to any inconsistency in the replicas as the database state is not modified. However, in case of a faulty coordinator, the application might have observed an erroneous database state during the course of the transaction, which might have led to the spurious decision of aborting the transaction. For example, consider a transaction trying to reserve a seat in a given flight with available seats. When the transaction queries the database for seat availability, a faulty coordinator might incorrectly return that no seats are available. As a consequence, the application program may decide to end the transaction with a ROLLBACK operation. If no verification of the results that were returned was performed, the client operation would have made a decision to rollback based on an incorrect database state.

To detect this, we decided to include an option to force the system to verify the correctness of the returned results also when a transaction ends with a ROLLBACK operation. When this option is selected, the execution of a rollback becomes similar to the execution of a commit (with the obvious difference that it is not necessary to check for write-write conflicts and that the transaction always aborts). If the verification fails, the ROLLBACK operation raises an exception.

4 Correctness

In this section we present a correctness argument for the design of Byzantium. We leave a formal correctness proof as future work.

Safety Our safety condition requires that transactions that are committed on the replicated database observe SI semantics.

Our correctness argument relies on the guarantees provided by the PBFT algorithm [3], namely that the PBFT replicated service is equivalent to a single, correct server that executes each operation sequentially. Since both the BEGIN and the COMMIT operations run as PBFT requests, this implies that every correct replica will observe the same state (in terms of which transactions have committed so far) both when they begin a

transaction and when they try to commit it. Furthermore, they decide on whether a transaction should commit or abort based on the sequence of values that clients observed (the same sequence is transmitted to all correct replicas as an argument to the PBFT request), and according to the SI semantics of their own local databases (whose state, as mentioned, is identical and reflects all transactions that have previously committed in the system). This implies that a correct replica will only allow a transaction to commit if it observed SI semantics (from the standpoint of this common database state) and therefore the outcome of the PBFT commit operation is also conforming to this semantics.

Liveness Under the same assumptions as PBFT, we guarantee that the BEGIN, COMMIT, and ABORT operations are eventually executed. Furthermore, operations that do not go through the PBFT protocol are simple RPCs which are live under the same set of assumptions. This guarantees the system makes progress.

5 Implementation

Deterministic behavior in database systems Our design requires deterministic behavior of operations, but some database operations are not deterministic (e.g., select). However, it is possible to force a deterministic behavior using some standard techniques (e.g., as used in [12, 13]).

Database locking issues When trying to commit a transaction in a replica, the transaction operations must be executed concurrently with other ongoing transactions (for which the replica is the primary replica). For database systems that use an optimistic concurrency control approach, this imposes no problems. However, for database systems that rely on locks, this can cause problems because executing a write operation requires obtaining a lock on the row that is being modified. However, some ongoing transaction could have already obtained the lock on that row for another write operation.

This problem is similar to the problem experienced by non-Byzantine replication systems that use snapshot isolation semantics and similar techniques can be used to address it (e.g., [5]) – either using write-sets or using widely available database operations for testing blocking behavior (*select ... for update nowait*). An ongoing transaction that would block the execution of the commitment process can then be aborted (this transaction would have to abort anyway due to a write-write conflict with the committing transaction).

6 Conclusion

This paper presented the design of Byzantium, a protocol for BFT database replication that provides SI semantics. Byzantium improves on the few examples of BFT databases by allowing for concurrent transaction processing, which is essential for performance, by not depending on any centralized components, on whose correctness the entire system relies, and using weaker semantics that allow greater concurrency. Byzantium takes advantage of the weaker SI semantics to avoid running every database operation through the expensive PBFT protocol, yet it serializes enough operations with respect to each other to guarantee this semantics.

We are currently completing our prototype and starting the evaluation of the system. In the future, we also intend to evaluate the overhead imposed by the use of a BFT replication algorithm as a black box, when compared with the use of a custom algorithm. We believe this aspect is rather important, as it will help us understand how useful BFT libraries can be for building complex services that tolerate Byzantine faults.

References

[1] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10. ACM Press, 1995.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[3] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186. USENIX Association, 1999.

[4] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proceedings of the 1st ACM EuroSys European Conference on Computer Systems 2006*, pages 117–130. ACM Press, 2006.

[5] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84. IEEE Computer Society, 2005.

[6] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[7] Hector Garcia-Molina, Frank M. Pittelli, and Susan B. Davidson. Applications of byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, 1986.

[8] Ilir Gashi, Peter T. Popov, Vladimir Stankovic, and Lorenzo Strigini. On designing dependable services with diverse off-the-shelf sql servers. In Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky, editors, *WADS*, volume 3069 of *Lecture Notes in Computer Science*, pages 191–214. Springer, 2003.

[9] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.

[10] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM Press, 2005.

[11] Christian Plattner and Gustavo Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174. Springer-Verlag New York, Inc., 2004.

[12] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 15–28. ACM Press, 2001.

[13] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 59–72, New York, NY, USA, 2007. ACM Press.