Efficient Middleware for Byzantine Fault Tolerant Database Replication

Rui Garcia
CITI / Departamento de Informática,
Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa
Quinta da Torre, Caparica, Portugal
bomgarcia@gmail.com

Rodrigo Rodrigues
MPI-SWS
Kaiserslautern and Saarbrücken,
Germany
rodrigo@mpi-sws.org

Nuno Preguiça
CITI / Departamento de Informática,
Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa
Quinta da Torre, Caparica, Portugal
nuno.preguica@di.fct.unl.pt

Abstract

Byzantine fault tolerance (BFT) enhances the reliability and availability of replicated systems subject to software bugs, malicious attacks, or other unexpected events. This paper presents Byzantium, a BFT database replication middleware that provides snapshot isolation semantics. It is the first BFT database system that allows for concurrent transaction execution without relying on a centralized component, which is essential for having both performance and robustness. Byzantium builds on an existing BFT library but extends it with a set of techniques for increasing concurrency in the execution of operations, for optimistically executing operations in a single replica, and for striping and load-balancing read operations across replicas. Experimental results show that our replication protocols introduce only a modest performance overhead for read-write dominated workloads and perform better than a non-replicated database system for read-only workloads.

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

General Terms Design, Performance, Reliability

Keywords Databases, Middleware, Byzantine Fault-tolerance.

1. Introduction

Database systems are a key component of the computer infrastructure of most organizations. It is thus crucial to ensure that database systems work correctly and continuously even in the presence of a variety of unexpected events. The key to ensuring high availability of database systems is to use replication. While many methods for database replication have been proposed [Cecchet 2008], most of these solutions only tolerate silent crashes of replicas, which occur when the system suffers hardware failures, power outages, etc.

While this approach suffices for many types of faults, it does not tolerate the effects of events such as software bugs or malicious attacks that can cause databases to fail in ways other than silently crashing. These types of events are of growing concern. Recent studies show that the majority of bugs reported for three commercial database systems would cause the system to fail in a non-crash manner [Gashi 2007, Vandiver 2007]; another study found that a significant fraction of concurrency bugs in MySQL led to subtle violations of database semantics [Fonseca 2010]. Intrusions have also been reported as being a problem: database systems have become a frequent target of attacks that can result in loss of data integrity or even permanent data loss [DISA 2004].

A promising approach for increasing the correctness and availability of systems in the face of these types of events is through Byzantine fault tolerant (BFT) replication. This class of replication protocols makes no assumptions about the behavior of faulty replicas (i.e., assumes a Byzantine fault model [Lamport 1982]), so it can tolerate arbitrary failures from a subset of its replicas (typically up to $\frac{1}{3}$ of the replicas).

However, the application of BFT techniques to database systems has been quite limited. Previous proposals either do not allow transactions to execute concurrently, which inherently limits the performance of the system [Garcia Molina 1986, Gashi 2007], or rely on a trusted coordinator node that chooses which requests to forward concurrently [Vandiver 2007]. In the latter case, the coordinator becomes a central point of failure: if the node crashes or is compromised the entire system becomes vulnerable.

In this paper we present Byzantium, a novel, middleware-based, Byzantine fault tolerant database replication solution. Byzantium improves on existing BFT replication for

EuroSys'11, April 10–13, 2011, Salzburg, Austria.
Copyright © 2011 ACM 978-1-4503-0634-8/11/04. . . $10.00

databases because it allows extensive concurrency with no centralized components (on whose correctness the integrity of the system depends), which is essential for achieving good performance and reliability.

Several design features in Byzantium are interesting in their own right, and several of them are useful beyond the scope of database replication. In particular, we highlight the following set of design choices and techniques that make the Byzantium design novel.

Snapshot isolation. Unlike previous BFT database systems, which provide 1-copy serializability [Bernstein 1986], Byzantium targets snapshot isolation semantics. We show how we can take advantage of these weaker semantics to increase concurrency, by restricting the use of a more expensive serialization protocol to a subset of the operations.

Middleware-based replication. This approach allows us to use existing database systems without modifying them, and even allows distinct implementations from different vendors to be used at different replicas. Such diversity is important in order to increase resilience against attacks triggered by software vulnerabilities or deterministic software bugs, but not having access to the database internals raises the bar for our protocols to achieve good performance.

Optimistic execution of groups of operations. Our system design proposes two alternative techniques for optimistically executing operations in a single replica. Each technique offers distinct advantages: one of them works transparently with any form of concurrency control, and the other offers better performance in the presence of Byzantine replicas, but requires extracting write-sets in databases that use locking. Furthermore, while optimistic execution takes advantage of transactional semantics, these techniques may be useful for replicating other types of systems, namely those that support some form of speculation [Nightingale 2005].

Striping with BFT replication. We also show how we can use BFT replication to improve the performance of operations that do not update the state of the system, by striping reads from different clients to different subsets of replicas, while maintaining the desired database semantics.

We implemented Byzantium and evaluated our prototype using variants of TPC-C. Our experimental results show that our replicated database has only a modest performance overhead for read-write dominated workloads and exhibits performance gains of up to 90% over executing transactions in a single replica for read-only workloads.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the system model and Section 4 details the system design and proposed algorithms. Section 5 argues the correctness of our solution. Section 6 discusses implementation issues and Section 7 presents an evaluation of our prototype. Section 8 concludes the paper.

2. Related work

We group the set of related proposals into two main areas: Byzantine fault tolerance and database replication.

Byzantine fault tolerance. Byzantine fault tolerant (BFT) replication has drawn a lot of attention in the last decade as a mechanism to mask malicious attacks and software errors. Over these years, there has been a series of proposals for replication protocols that tolerate Byzantine faults and claim to be efficient enough to be practical (e.g., PBFT [Castro 2002], or Zyzzyva [Kotla 2007]). Most such proposals have centered around building a *replicated state machine* [Schneider 1990]. These protocols allow the replication of a deterministic service that follows an RPC model: clients issue requests, servers maintain the state of the service, and given a certain request and current state, the servers deterministically issue a reply and move on to the subsequent state. These protocols have been used to provide Byzantine fault tolerance to different services, including distributed file systems and coordination services [Castro 2002, Clement 2009].

While these protocols could also be used to replicate a database server by executing individual database operations as operations of the replicated state machine, this would preclude operations from being executed concurrently, thus limiting performance. Furthermore, for database systems using lock-based concurrency control, an operation that blocked would prevent any further operations from being executed.

We build on these proposals since our system uses state machine BFT replication as one of its building blocks. However, we address these performance limitations by taking advantage of snapshot isolation semantics and optimistic execution, which allow us to craft our protocols in a way that avoids the use of state machine replication when possible, thus increasing concurrency.

Database replication. There exist several proposals, both from industry and academia, for replicating database systems in order to both increase their resilience to faults and improve their throughput (see [Cecchet 2008] for an overview of the topic).

The proposals that are more closely related to our work are the various proposals for middleware-based replication that provide snapshot isolation (SI) [Elnikety 2005, Lin 2005, Plattner 2004]. In particular, one of the two alternative protocols we present uses an approach that bears similarity to Ganymed [Plattner 2004], with all read-write transactions executing first in the same replica responsible for guaranteeing SI properties. The execution of read-only transactions is distributed among the other replicas. The other alternative protocol we present bears similarity to the approach proposed by Elnikety et. al. [Elnikety 2005], with read-write transactions executing first in a single replica (possibly using different replicas for concurrent transactions) and, at commit time, transactions being propagated to the other replicas that detect conflicts according to the SI properties. While Byzantium uses similar techniques, our solution differs in that it is designed with the assumption of a Byzantine fault model, instead of assuming that nodes fail only by crashing.

There have also been a few proposals for Byzantine fault tolerant database replication.

The initial protocols in this area were proposed by Garcia-Molina et al. [Garcia Molina 1986] and by Gashi et al. [Gashi 2007]. Their replication protocols serialize all requests and do not allow transactions to execute concurrently. Consequently, their protocols provide stronger semantics than ours, but serializing all requests inherently limits the performance of the system. We improve on these systems by using weaker semantics and novel protocols to obtain better performance.

HRDB [Vandiver 2007] provides BFT database replication using a trusted node to coordinate 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, the coordinator is a single point of failure: if the coordinator crashes, the availability of the system will be affected until it recovers. Finally, HRDB ensures 1-copy serializability, whereas our approach provides weaker (yet commonly used) semantics in order to achieve good performance.

In a prior workshop paper we described a preliminary design of the multi-master version [Preguiça 2008]. This paper improves on that work in several ways. First, we modified the design to accommodate changes that were necessary to obtain good performance (many of which were driven by observing the performance limitations of the original design in a real deployment). In particular, we have optimized the execution of read-only transactions to allow them to execute in a small subset of the replicas, and we changed the way we execute operations so that they are propagated to all replicas before commit time. Second, we propose a second version of the system, based on a single-master approach. This approach requires less support from the underlying database system. Finally, we present a complete implementation and an experimental evaluation of our prototype.

3. System model

We assume a Byzantine failure model where faulty nodes (client or servers) may behave arbitrarily, other than not being able to break the cryptographic techniques that are used. We assume at most f replicas are faulty out of a total of $n = 3f + 1$ replicas. In our current implementation we do not employ existing proactive recovery mechanisms [Castro 2002], which implies that we need to ensure that we have no more than f faulty nodes throughout the system lifetime. If we were to apply proactive recovery techniques, we would be able to tolerate at most f faults during a window of vulnerability.

When the correctness conditions of the system are met, the safety property ensured by Byzantium is that the replicated database provides ACID semantics, with *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 is an attractive level of isolation for several reasons: it allows for increased concurrency among transactions when compared to stronger properties such as 1-copy serializability, it is implemented by many commercial database systems, it provides identical results to 1-copy serializability for many typical workloads (including the most widely used database benchmarks, TPC-A, TPC-B, TPC-C, and TPC-W) [Elnikety 2006], and there exist techniques to transform a general application program so that its execution under SI is equivalent to strict serializability [Fekete 2005].

Our system guarantees these safety properties in an asynchronous distributed system: we assume nodes are connected by an unreliable network that may fail to deliver messages, corrupt them, delay them arbitrarily, or deliver them out of order. Many database replication protocols make stronger assumptions about the timely delivery of network messages, so they can use timeouts to detect replica faults. This assumption can be problematic because when it is not met safety violations may occur. To give a simple example, in a primary-backup scheme, two machines can believe erroneously that the other has failed. This could lead to the existence of two primaries that accept new updates independently, leading to state divergence. Furthermore, this assumption can be broken either by deliberate attacks (e.g., by flooding a correct node) or by other occurrences like longer than usual garbage collection cycles that lead to increases in message processing delays [Gribble 2001].

The fact that we are ensuring safety in an asynchronous network model implies that we need to assume some form of synchrony for liveness [Fischer 1985]. Thus our system only guarantees that clients can make progress during periods when the delay to deliver a message is bounded. This assumption is referred to as eventual synchrony, and is a common assumption for liveness in replicated systems that ensure safety despite asynchrony [Castro 2002].

Note that the goal of Byzantium is to ensure correctness and high availability of the system despite arbitrary faults, and not to defend against attacks that try to violate the confidentiality of database contents. There exist, however, extensions to BFT protocols to address the problem of confidentiality [Yin 2003] that we could leverage in our work.

3.1 Database and BFT protocol requirements

Our system employs two components that are used as black boxes, but are required to provide certain semantics.

The first component is a BFT state machine replication protocol. This protocol must implement linearizable semantics [Herlihy 1987], which is the case for most BFT state machine replication proposals [Castro 2002, Kotla 2007]. In Byzantium, we used a BFT system that requires $3f+1$ replicas, but as future work we would like to modify the implementation and evaluate the performance change when using a BFT system that only requires $2f+1$ execution replicas [Yin 2003]. This would bring advantages in terms of the aggregate machine load and the inter-replica communication costs, but also provide fewer opportunities for using striping to improve the performance of read-only transactions (which dominate several common workloads).

The other component is an off-the-shelf database system, for which we consider a standard model where 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. We require that the off-the-shelf database provides snapshot isolation semantics and supports savepoints (both of which are common in database systems).

While we do not place restrictions on the concurrency control mechanism the databases implement, one of the alternative designs we propose requires support for extracting write-sets in databases that use locking. As in other (non-BFT) replicated databases (e.g., [Elnikety 2006]) we extract write-sets from ongoing transactions using database triggers.

4. System design

In this section we present the design and algorithms used in the Byzantium system.

4.1 Architecture

Byzantium is built as a middleware system with the architecture depicted in Figure 1. The system is composed of a set of $n = 3f+1$ server replicas and a finite number of clients.

Each replica is composed of a database system and the Byzantium replica proxy, which is linked with both a communication library and the replica-side BFT library. The communication library allows for the client to communicate directly with the replicas without going through the more expensive serialization provided by the BFT replication protocol. The communication library implements a light-weight retransmission protocol to ensure that messages are delivered in FIFO order, but, unlike BFT operations, these messages can reach different replicas in different orders. Each replica maintains a full copy of the data in an off-the-shelf database system; i.e., we use a shared-nothing architecture (to ensure fault isolation with Byzantine nodes) and we do not partition data. The replica proxy is responsible for handling client requests and controlling the execution of operations in the database system, guaranteeing that the operations execute with the desired semantics in all non-faulty replicas, and that, when the system quiesces, the state of the database in all non-faulty replicas is the same.

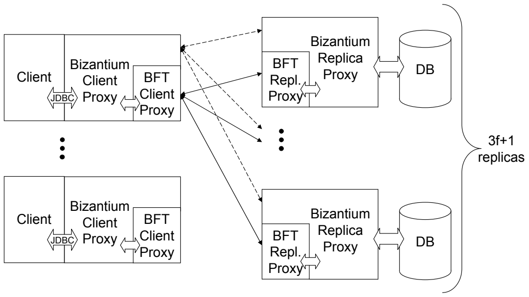

Figure 1. System Architecture.

User applications run on client nodes and access our system using the JDBC interface. Thus, applications that access conventional database systems using a JDBC interface can use Byzantium without 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 BFT state machine replication protocol, and therefore this proxy is linked with the client side of the BFT replication library.

As mentioned, two of the components of the architecture, namely the database system and the BFT replication library, are used as black boxes. (In some cases, where write-set extraction is required, the approach may more accurately be termed "gray-box".)

Using an off-the-shelf, black-box database provides several advantages: it enables the use of third party databases that may not support replication and whose source code may not be accessible, it allows for upgrading the database server without having to update the replication code, and it allows for different replicas to run different implementations of the database server. The latter capability is important for ensuring a lower degree of fault correlation, in particular when these faults are caused by deterministic software bugs [Rodrigues 2001, Vandiver 2007]. Running distinct versions is facilitated by the fact that replicas use the JDBC interface to communicate with the database system. Thus we can easily swap between database servers that implement this interface. By default, we configured our prototype implementation to run an instance of PostgreSQL at each replica.

The other black-box component is the BFT replication library. Byzantium uses an implementation of the PBFT protocol [Castro 2002]. The PBFT library provides two main interfaces: on the client side, the library offers a "BFT_invoke" method that issues the request and returns the corresponding reply, implementing the client side of the protocol; on the server-side, the library executes the replica protocol for serializing all requests, and once a request is serialized it invokes a "BFT_execute" upcall, whereby the application-specific code executes the client request, updates the service state, and produces the corresponding reply. Note that these libraries only execute one request at a time, since they must assume that the state changes performed by a given request may affect the outcome of the next one. Since we use PBFT as a black box, we can easily switch this replication library with a different one, provided it offers the same guarantees (i.e., state machine replication with linearizable semantics) and has a similar programming interface.

4.2 Normal case operation

We start by describing the normal case in which clients and replicas are not Byzantine, and thus all nodes follow the protocols we present. We address the cases of malicious behavior in subsequent sections. 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. For simplicity, the code omits some details such as error handling, message signing and optimizations. This code is used to implement two versions of the system, termed single-master and multi-master. The difference between the two versions, as far as the code in Figure 3 goes, lies only in the selection of replicas to be contacted, as explained later.

At a high-level, our approach is to only force a total order among the operations for which doing so is required to ensure that all transactions execute against the same snapshot at all replicas, i.e., the BEGIN and COMMIT/ROLLBACK operations. For these, we rely on the PBFT protocol to enforce a total order at all replicas despite Byzantine faults. The remaining operations (reads and writes) can be executed more efficiently by propagating them using the unreliable multicast mechanism, and executing them in a single replica, concurrently with other operations. The main problem then becomes how to deal with the case when that replica is Byzantine and returns wrong results. This is handled at commit time by validating the read and write results at all replicas.

In more detail, the application program starts a transaction by executing a BEGIN call on the database interface (*function db_begin*, Figure 2, line 1). The client starts by generating a unique identifier for the transaction and selecting one replica to speculatively execute the transaction – we call this replica the master replica for the transaction. (Note that it does not need to be the same primary replica that is used by the PBFT protocol.) Then, the client issues the corresponding PBFT operation that will serialize the transaction begin in all replicas by calling the *BFT_invoke(<BEGIN,...>)* method from the PBFT library. The execution of the protocol will eventually trigger the corresponding *BFT_execute* upcall (Figure 3, line 1) at all replicas. At that moment, a database transaction is started at the replica. Given the properties of the PBFT system, and since both BEGIN and COMMIT operations execute serially as PBFT operations, 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 (functions *db_read_op* and *db_write_op*, Figure 2, line 11 and line 24 respectively). In a write operation, the operation is multicast to all replicas (by calling *mcast*, which triggers the corresponding *receive* upcall in all replicas, shown in Figure 3, line 33). The operation is received by all replicas but only executed in the master and its result is returned to the client, who then returns the result to the application. Both the client proxy and the master replica keep a list of operations and their results.

Read operations can be executed in two different ways. If the transaction is known to be read-write (i.e., after the first write operation), the execution of a read operation is similar to the execution of a write operation. Otherwise, we perform an optimized read as described in Section 4.4.

The transaction concludes by executing a COMMIT operation (*function db_commit*, Figure 2, line 32). At commit time, it is necessary to (1) serialize all commit operations among themselves and relative to snapshots for beginning transactions, and (2) perform a series of validations, namely to confirm the outputs of the read and write operations that were executed in a single (potentially Byzantine) replica. To achieve this, the client issues the COMMIT PBFT operation that includes the digest of the operations that were issued and their results. This will trigger the *BFT_execute* upcall at all replicas (Figure 3, line 9), and the invocation of this upcall will occur in the same order relative to other BEGIN and COMMIT operations.

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 that arise with this approach when the master is Byzantine in Section 4.6.

4.3 Optimistic execution and recovery

As we explained, the read and write operations of a transaction execute optimistically in a single replica, and their results need to be subsequently validated at commit time. We developed two algorithms that differ in how the master replica is chosen and how operations execute (or not) optimistically in non-master replicas. While the single-master technique performs better for read-write dominated workloads, the multi-master version performs better when there is a large number of read-only transactions. Additionally, these techniques imply a trade-off between the support required from the database system and the performance in the presence of a Byzantine master. We discuss each of these techniques in turn.

Multi-master optimistic execution. In the multi-master version, each client can select a different replica as the mas-

```
function db_begin() : txHandle
    uid = generate new uid
    (masterRep,readReps)=select(1 replica,f+1 replicas)
    BFT_invoke(<BEGIN,uid,(masterRep,readReps)>)
    ops = new Map
    readOnly = true
    opCount = 0
    trxHandle = new trxHandle(uid, masterRep, readReps, opCount, ops, readOnly)
    return txHandle

function db_read_op(txHandle, op) : result
    opNum = ++txHandle.opCount
    mcast(<read(txHandle.uid, opNum, op)>)
    recv(<readResult(txHandle.uid,opNum,HOp,res)>          // first result
    txHandle.ops.put(opNum,<op,1,H(res),'read'>);
    return res
background                                                 // additional results
    recv(<readResult(txHandle.uid, opNum,HOp,res)>)
    <_,count,HRes,_> = txHandle.ops.get(opNum)
    if(HRes == H(res)) then
        trxHandle.ops.put(opNum,<op,count+1,HRes,'read'>);
    endif

function db_write_op(txHandle, op) : result
    opNum = ++txHandle.opCount
    mcast(<write(txHandle.uid, opNum, op)>)
    recv(<writeResult(txHandle.uid,opNum,HOp,res)>)
    txHandle.readOnly = false
    txHandle.ops.put(opNum, <op,1,H(res),'write'>);
    return res

function db_commit(txHandle)
    concurrent
        if(txHandle.readOnly) then
            while LastReadConfirmed(txHandle)<txHandle.opCount
                wait
            end while
            return
        endif
    with
        lastConfirmed=LastReadConfirmed(txHandle)
        HOps = H(ListOps(txHandle))
        HRes = H(ListRes(txHandle, lastConfirmed))
        res = BFT_invoke(<COMMIT, txHandle.uid, lastConfirmed,HOps,HRes>)
        if(res == true) then
            return
        else
            throw ByzantineExecutionException
        endif
    end concurrent
```

Figure 2. Byzantium client proxy code.

```
upcall for BFT_exec(<BEGIN,uid,(masterRep,readReps)>)
    DBTxHandle = db.begin()
    ops = new Map
    readOnly = true
    txSrvHandle = new txSrvHandle(uid, DBTxHandle, masterRep, readReps,
                                  ops, readOnly)
    openTxs.put(uid, trxSrvHandle)

upcall for BFT_exec(<COMMIT, uid, lastConfirmed, cltHOps, cltHRes>) : boolean
    txSrvHandle = openTxs.get(uid)
    openTxs.remove(uid)
    if( NOT ThisIsMasterReplica( txSrvHandle)) then
        execOK = exec_and_verify(txSrvHandle.DBTxHandle, lastConfirmed,
                                 txSrvHandle.ops, cltHOps, cltHRes)
        if(NOT execOK) then
            DBTxHandle.rollback()
            return false
        endif
    endif
    return DB_trx_handle.commit()

upcall for recv(<read(uid, opNum, op)>)
    txSrvHandle = openTxs.get(uid)
    txSrvHandle.ops.put(opNum, <op,->)
    if((txSrvHandle.readOnly AND ThisIsReadReplica(txSrvHandle))
            OR( NOT txSrvHandle.readOnly AND
                        ThisIsMasterReplica( txSrvHandle))) then
        result = txSrvHandle.DBTxHandle.exec(op)
        txSrvHandle.ops.put(opNum, <op,result>);
        send_reply(<readResult(uid,opNum,H(op),result)>)
    endif

upcall for recv(<write(uid, opNum, op)>)
    txSrvHandle = openTxs.get(uid)
    txSrvHandle.ops.put(opNum, <op,->)
    if(txSrvHandle.masterRep == THIS_REPLICA) then
        if((txSrvHandle.readOnly AND NOT ThisIsReadReplica( txSrvHandle)) then
                ExecReadPrefix( trxHandle.ops)
        endif
        result = txSrvHandle.DBTxHandle.exec( op)
        txSrvHandle.ops.put( opNum, <op,result>)
        send_reply(<writeResult(uid,opNum,H(op),result)>)
    endif
    txSrvHandle.readOnly = false
```

Figure 3. Byzantium replica proxy code.

ter. This leads to more flexibility in terms of load balancing, and good resilience to a Byzantine master.

In this version of the protocol, the master is selected in the beginning of the transaction either at random or following a more sophisticated load-balancing scheme [Elnikety 2007]. Subsequent reads and writes are then performed optimistically at the master replica, which, as we pointed out, may be Byzantine and return incorrect results. Therefore, when the transaction attempts to commit there are two validation steps that need to be performed: ensuring that the results the client observed were correct, and that the transaction can commit according to SI.

For the former, all non-master replicas have to execute the remaining transaction operations and verify that the returned results match the results previously output by the master (Figure 3 line 12). (For now we assume that these operations were received by all replicas, and in Section 4.5.2 we explain how to handle the case where replicas do not receive them, either due to message loss or to a Byzantine client.) Since the transaction executes in the same snapshot in every replica (due to the fact that both BEGIN and COMMIT operations are serialized by PBFT), if the master and client were correct, all other correct replicas should obtain the same results. In case the results do not match, either the client or the master was Byzantine and we rollback the transaction.

Additionally, all replicas including the master must guarantee that SI properties hold for the committing transaction. The way this is done depends on the concurrency control mechanism of the underlying database system.

For database systems with optimistic concurrency control, guaranteeing SI is immediate. As a transaction executes in the same snapshot and commits in the same order relative to other commits in all replicas, the validation performed by the database system suffices to guarantee that a transaction commit succeeds in all replicas or in none (when a conflicting transaction has previously committed). With these database systems, all operations can be executed optimistically in all replicas, reducing the work needed at commit.

However, most database systems rely on locks for concurrency control, including the main system we used to test our prototype, PostgreSQL. In such systems, before executing a write operation, the transaction must obtain a lock on rows to be written. When a replica is acting as master for some ongoing transaction, it will obtain locks on rows it changes. This can block the local execution of a committing transaction that has written in the same rows, and ultimately lead to a deadlock. We address this problem by temporarily undoing all operations of the ongoing transaction. After executing the committing transaction, if the commit succeeds, the ongoing transaction is rolled back due to a conflict. Otherwise, we replay the undone operations and the ongoing transaction execution may proceed.

To achieve this, we rely on the widely available transaction *savepoint* mechanism. Savepoints enable rolling back all operations executed inside a running transaction after the savepoint is established. Thus, when the BEGIN operation executes, a savepoint is created in the initial database snapshot. Later, when we need to undo the operations that were executed in the transaction but still use the same database snapshot, we just need to rollback the transaction to the savepoint that was previously created.

To know which local transaction would block a committing transaction, we use approaches similar to non-BFT replicated databases with SI semantics [Elnikety 2006] - we further discuss this issue in Section 6.

Single-master optimistic execution. Unlike in the multi-master protocol, where we have to extract write-sets in databases the use locking, in the single-master case we can avoid this by optimistically executing transactions in the same single node. This requires that clients and replicas agree on the single master that should be executing reads and writes in the course of transactions. This is achieved by augmenting the service state maintained by the PBFT service with this information, and augmenting the service interface with special operations to enable changing the current master. The properties of PBFT will then ensure that all clients and replicas agree on which replica is acting as a master, and this can be communicated to the client as part of the output of the BEGIN PBFT operation.

Given this agreement, validating the SI properties can be done in a straightforward manner just by executing read and write operations of each transaction when it commits. In this case, as transactions commit serially in all replicas, if the transaction can commit in the master, it will be able to commit in all non-master replicas independently of the concurrency control mechanism that is used.

This scheme can be extended to allow transactions to also execute speculatively in non-master replicas before commit time. In databases using optimistic concurrency control, operations can be broadcast to all replicas, which speculatively execute them as they are received. However, for this optimization to work in lock-based database systems, we must guarantee that the same transactions obtain the same set of locks in all replicas – otherwise, some committing transaction would not be able to obtain the needed locks. Guaranteeing this without controlling the internals of the database system requires guaranteeing that operations are issued in some given order to the database system. To ensure this, HRDB [Vandiver 2007] proposes the use of commit barriers controlled by a centralized coordinator for this purpose. In our system we implement a similar idea of using information provided by the master to coordinate other replicas, guaranteeing that all operations of a transaction but the last one can execute speculatively in non-master replicas before commit-time. However, unlike HRDB, if the master is suspected to be faulty, another replica is selected to act as primary.

In particular, our approach leverages the fact that, for transactions executing speculatively at non-master replicas, if an operation completes in the master, it can execute in other replicas without blocking. This follows from the fact that any other operation that would require the same locks would have blocked in the master. Thus, when a non-master replica receives an operation op_n from t_1, it knows that it can execute operation op_{n-1} from transaction t_1 (because op_{n-1} has not blocked in the master). However, this condition is not sufficient to avoid blocking when a transaction commits – e.g. suppose an operation op of t_2 executes at the master after the master committed t_1. If a non-master replica executes

op before running some operation of t_1 that requires the same locks as op, t_1 would be unable to commit. To address this problem, we must guarantee that if an operation was executed after the commit of t_n at the master replica, that same operation will execute in all non-master replicas after committing t_n (this was known as the *Transaction-ordering rule* in HRDB). To guarantee this, the message with the result of a write operation, op_n, includes the number of the last committed transaction, t_m, at the master (serialized by the PBFT protocol). The message that propagates a read or write operation op_{n+1}, includes the value received in the result of the last write operation. Therefore, we can enforce the necessary order by imposing that operation op_n executes at non-master replicas only after that replica has executed the commit of t_m.

4.4 Read-only optimizations

When transactions begin, we assume they are read-only until the first write operation. While the transaction is flagged as read-only, we employ the following optimization to improve the performance of read-only transactions (and of read-only prefixes of other transactions). Read requests are executed in $f+1$ replicas (chosen randomly when the transaction begins) and the result from the first reply is returned to the client, while the remaining replies are collected in the background (Figure 2, lines 17-22). When the f additional replies that are received in the background match the first reply, the result of the read is considered to be confirmed without the need for executing the operation in any additional replica.

At commit time, if all returned values were correctly validated by $f + 1$ replicas, the client immediately returns the commit successfully. In the case that some reads were not yet validated, the commit procedure falls back to the original, unoptimized version, which is run in parallel with trying to conclude the optimized validation. In the normal case, when the $f+1$ replicas reply at similar speeds, operations are confirmed by the optimized protocol. When a write operation occurs, the transaction is promoted to read-write, and starts executing the normal protocol. However, the confirmed read operations executed prior to the first write will not be included in the final commit-time validation.

This scheme enables a form of striping and load-balancing, since read-only transactions only execute their read operations in $f + 1$ of the $3f + 1$ replicas. For providing efficient load balancing, the selection of the replicas that execute the reads (Figure 2, line 3) could follow one of the various existing proposals (e.g., [Elnikety 2007]). In our prototype, the $f+1$ replicas are selected randomly with the constraint that, in the multi-master version, the $f+1$ replicas always include the master. This has the advantage that, when a transaction is upgraded to read-write, the master replica has already executed all previous operations and is ready to proceed with the execution of the subsequent operations. In the single-master version, the $f + 1$ replicas selected to execute read operations do not include the master (as in Ganymed [Plattner 2004]). The rationale for this approach is to reduce the load in the master node – otherwise, the master node would have to execute all transactions. The downside of this approach is that when a transaction is upgraded to read-write, the master node needs to execute all operations whose results are not known to be guaranteed, if any.

4.5 Tolerating Byzantine faults

So far we have mostly assumed that nodes follow the protocol. In this section we explain how the system handles Byzantine behavior, starting with the assumption that only replicas may exhibit Byzantine behavior, and later addressing the case of Byzantine clients.

4.5.1 Tolerating a faulty master

A faulty master can return erroneous results or fail to return any results to the clients. The first situation is addressed by having all replicas verify at commit time the correctness of results returned by the master. If the results of executing the operations in the transaction do not match the results that the client observed (and whose digests are passed as an argument to the PBFT COMMIT operation), the replicas will rollback the transaction and the client will throw an exception signaling Byzantine behavior. This guarantees that correct replicas will only commit transactions for which the master has returned correct results for every operation.

Addressing the case where a master fails to reply to an operation requires different solutions, depending on whether a single-master or a multi-master approach is used.

Multi-master In the multi-master version, if the master fails to reply to an operation, the client selects a new master to replace the previous one and starts by re-executing all previously executed transaction operations in the new master. If the obtained results do not match, the client rollbacks the transaction by executing a ROLLBACK operation and throws an exception signaling Byzantine behavior. If the results match, the client proceeds by executing the next operation in the new master.

Because of this mechanism, there may be situations where the master is unaware that it did not execute the entire transaction (e.g., if the client switched to a new master due to temporary unreachability of the original master). To handle this, at commit time, a replica that believes itself to be the master of a transaction must still verify that the sequence of operations sent by the client is the same as the sequence that it has itself executed. Thus, if the master that was replaced is active, it will find out that additional operations exist and will execute them.

In subsequent transactions, a client will not select as master a replica that it suspects of exhibiting Byzantine behavior.

Single-master In this scheme, if the master fails to reply to an operation, the client will forward the request to all replicas. Each replica will try to forward the request to the master on behalf of the client. If the master replies, the replica will forward the reply to the client. Otherwise, the replica will suspect the master, and request a master change.

A replica starts a master change by submitting a PBFT *master change* operation (as a consequence of a suspicion of Byzantine behavior). When $f + 1$ *master change* operations from different replicas are executed concerning the same master, a new master is automatically elected – in our prototype, replicas are numbered, and the next master replica is selected in a round-robin fashion. In this case, all ongoing transactions are marked for rollback. When executing the next operation, the client will be informed that the transaction will rollback.

Aside from this situation, a replica will also request a master change when, during the execution of the commit, the results observed by the client do not match the local results. This raises the possibility of Byzantine clients using this mechanism to cause false positives and trigger constant master changes. The next section discusses this and other avenues that Byzantine clients may use to cause the system to malfunction.

4.5.2 Tolerating Byzantine clients

The system also 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 with legitimate access from writing incorrect data, or deleting entries in the database. Such attacks can only be limited by enforcing security/access control policies and maintaining database snapshots that can be used for data recovery [King 2005]. What we are trying to prevent are violations of database semantics or service availability due to clients that do not follow the protocol we described.

Since PBFT already ensures that Byzantine clients cannot affect the system other than by invoking operations through the service interface, our system only needs to address violations of the remaining protocols that are used.

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

There are, however, more subtle deviations that could be exploited by Byzantine clients. One avenue of attack follows from that fact that during a transaction operations are multicast to all replicas, and, at commit time, the client propagates a digest of operations and results to all replicas, but not the operations themselves. A Byzantine client could exploit this behavior by sending different sets of operations to different replicas. (A similar possibility is that some of the messages containing operations are lost and do not reach some of the replicas by commit time.) The consequence would be that at commit time, only those replicas that had a sequence that matched the digests would commit the transaction, leading to divergent database states at different replicas.

To address this problem, while avoiding a new round of messages among replicas during correct executions, we leverage a PBFT protocol mechanism that enables replicas to agree on non-deterministic choices [Rodrigues 2001]. This feature was originally used for replicas to agree on things such as the current clock value. In this mechanism, the primary proposes a value for the non-deterministic choices and replicas that disagree with that value can reject it. If there is no set of $2f + 1$ replicas that accept that choice then the operation is not executed, a primary change will take place, and the new primary can then propose a different value.

We use this mechanism to allow replicas to vote on whether they have the sequence of operations that match the digests sent by the client. If $2f + 1$ replicas agree on the fact that they hold all the operations, the PBFT operation will proceed with transaction commitment. Correct replicas that were not in this set and do not have the correct sequence of operations must obtain it from other replicas. If the primary believes it does not have the sequence of operations and $2f + 1$ replicas agree on this fact (if the client sent incorrect digests, for instance) then the PBFT operation proceeds with all replicas rolling back the transaction. Otherwise, if there is no agreement among any set of $2f + 1$ nodes, the PBFT protocol automatically initiates a primary change and the new primary will repeat the process. In parallel, correct replicas that have the right sequence of operations will multicast them to all replicas. This ensures liveness, since either a correct replica has the set of operations and will eventually propagate them to all replicas, or the correct replicas will eventually agree on the fact that they do not have access to the operations and the transaction will rollback.

Another possible consequence of Byzantine clients is that the master could be forced to discard its previously executed sequence of operations. This would be the case if the set of values sent in the commit operation is accepted by $2f + 1$ replicas but these values do not correspond to the values sent to the master before committing. In this case we need to allow the master to undo the executed operations and execute the new sequence in the original snapshot. To this end, we set up a savepoint when the transaction starts. Later, if the master finds that the Byzantine client had sent other replicas a different set of values that match the commit digests, then the transaction is rolled back to the savepoint that was previously created before executing the new sequence of operations. This ensures that all replicas, including the master, execute the same operations on the same database snapshot, guaranteeing the correct behavior of our system.

Another possible point of exploitation arises if a Byzantine client send an incorrect digest for the results, leading all replicas but the master to rollback the transaction. To address this case, the master checks the received digest and rollbacks the transaction if a Byzantine behavior is detected.

Finally, to avoid the aforementioned problem of clients constantly changing the master due to false accusations in the single master approach, we can deploy a mechanism by which replicas suspect a client that causes too many master changes, and that client is forced to ask for a receipt, signed by the master replica, of the operation results that were returned to it before being able to cause a new master change. The operation to request a receipt can be handled just like read and write operations.

4.6 Handling rolled back transactions

When a transaction ends with a ROLLBACK operation, a possible approach is to simply rollback the transaction in all replicas without verifying if previously returned results were correct (e.g., this solution is adopted in [Vandiver 2007]). In our system, this could be easily implemented by executing a PBFT operation that rollbacks the transaction in each replica.

This approach does not lead to any inconsistency as the replicas are not modified. However, in case of a faulty master, the application might have received an erroneous result, leading to the decision to rollback the transaction. For example, consider a transaction trying to reserve a seat in a flight that has seats available. When the transaction queries the database for available seats, a Byzantine master might incorrectly return that none is available. As a consequence, the application program may end the transaction with a ROLLBACK. If no verification of the results that were returned was performed, the client would have made a decision to rollback the transaction 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 even when a transaction ends with a ROLLBACK operation. When this option is activated, the execution of a rollback becomes similar to the execution of a commit (with the obvious difference that the transaction always rollbacks). If the verification fails, the ROLLBACK operation raises an exception. Note that a correct program should include the code to catch all exceptions raised by a database operation and take appropriate action depending on the content of the exception.

5. Correctness

In this section we sketch a proof that our design meets safety and liveness conditions.

The safety part of our correctness conditions states that transactions that are committed on the replicated database observe SI semantics. This follows from the linearizable semantics of PBFT [Castro 2002], and the fact that all BEGIN and COMMIT operations are serialized by PBFT and thus execute in the same total order at all non-faulty replicas. This, coupled with the fact that the output of the commit operation only depends on the sequence of begin and commit operations that happened previously, which is the same at all non-faulty replicas, implies that the output of commits will be the same as the output of the local commit at each non-faulty replica. Note that the output of commit is independent of the other operations that are not serialized through PBFT because commit operations carry as argument the sequence of values that were read and written by the transaction.

Given this point, the proof that the system obeys SI semantics follows from the fact that each non-faulty replica applies the begin and commit operations on their local database that provides SI semantics and forwards the reply from the database, and that the commit validates all the outputs that the client received and applies all the updates that the client issued during the transaction.

Read-only transactions are a special case since the commit does not require invoking a PBFT operation, but these also conform to SI semantics, since when these transactions begin they run a PBFT begin operation, hence establishing a position for this transaction in the serial order of committed transactions at all non-faulty replicas, as stated above. Since the values that are read are confirmed by $f+1$ replicas, there is at least one non-faulty replica in that set that will return a value that is correct according to the SI semantics and the total order set by the PBFT begin operation.

For liveness, we need to ensure that operations that are initiated by the client are eventually executed. This requires the same assumptions that are required for liveness of the PBFT protocol, which is that after some unknown point in the execution of the program messages message delays do not grow superlinearly [Castro 2002].

Given this assumption, and due to PBFT's liveness condition, we can guarantee that the BEGIN, COMMIT, and ROLLBACK operations are eventually executed. Furthermore, operations that do not go through the PBFT protocol are simple RPCs which are eventually executed under the same set of assumptions. The execution of these operations does not block, by algorithm construction, and thus we can guarantee that all client operations eventually get executed.

6. Implementation

We implemented a prototype of Byzantium in Java. We developed a Java-based PBFT implementation, and our proxies use the JDBC interface to interact with the application and the underlying database system. We also built a communication library providing FIFO semantics and message authentication using Message Authentication Codes (MACs).

We use several techniques proposed in other middleware database replication systems [Cecchet 2008, Elnikety 2006, Rodrigues 2001, Vandiver 2007] in our system. We make non-deterministic database operations (e.g. select) deterministic by rewriting the operation and/or overwriting the non-deterministic components of each reply.

We implemented two mechanisms to avoid deadlocking in our multi-master version: one that relies on the extraction of write-sets using database triggers, and another that relies on the analysis of SQL code. These mechanisms work by maintaining the write-sets of on-going transactions and, before executing a remote operation, verify if it would conflict with local transaction. Sometimes it is not possible to verify this from the write code – in such situations, it is possible to obtain the additional needed information by relying on the SELECT ... FOR UPDATE NOWAIT SQL statement. Our experiments have shown that neither approach offers a clear performance advantage over the other.

Since Byzantium was designed to work with any database that supports the JDBC interface and provides Snapshot Isolation, we tried it with two different database implementations: PostgreSQL and HyperSQL[1], a Java-based database system that implements SI. In HyperSQL, the single-master version worked without changes, but for the multi-master version it was necessary to develop a workaround to address an unimplemented method of the JDBC interface. The performance of HyperSQL with the TPC-C benchmark configuration we used was much lower than that of PostgreSQL. Thus, in our evaluation, we only report the performance with PostgreSQL.

7. Evaluation

In this section, we evaluate the performance of our prototype. Our tests were performed with $f = 1$, leading to $n = 3f+1 = 4$ replicas. Studies show that this configuration is sufficient to mask almost all reported database bugs [Gashi 2007]. The tests were run on a cluster of machines, each one with a single-core 2.6 GHz AMD Opeteron 252 processor, 4 GB of memory, a 146 GB Ultra320 SCSI disk and 1 Gigabit ethernet ports. The machines were connected by a Nortel Ethernet Routing Switch 5520. The machines were running the Linux operating system, kernel version 2.6.30. The database used was PostgreSQL version 8.3.4, in synchronous commit mode to guarantee reliability. The JVM we used was Sun VM version 1.6.0_12.

The evaluation used an open-source implementation of TPC-C[2]. We made slight modifications to the benchmark, namely to include a warm-up phase before starting the performance measurements, and to allow clients to execute on different machines. An important point is that our benchmark does not use database batches, which makes it more demanding for the communication protocols and leads to worse performance when compared to the solution using batches. Our database configuration included 10 warehouses and an inter-transaction arrival time of 200 ms. The experiments show the average of 5 runs, with the error bars showing the lowest and highest value.

The goal of our experiments is to evaluate the overhead of providing BFT replication and the efficiency of the two versions of our protocols. For this, we compare the Byzantium multi-master, *Byz-N*, and single master, *Byz-1*, versions to both a non-replicated *proxy*-based solution and a *full BFT* replicated system where all operations are serialized by executing each operation as a PBFT operation. The *proxy* solution uses a proxy that relays the connections from all clients to a single database server using JDBC connections. The solution is multi-threaded, with each client being handled concurrently by a dedicated thread. This reflects the performance of a non-replicated database server, while only adding the overhead of implementing a Java-based middleware solution. We use the *proxy* and *full BFT* solutions as comparison points, because they represent the best and worst case of what an implementation using our code base is expected to achieve, with the former incurring no BFT overhead and the latter incurring the overhead of running PBFT for every operation.

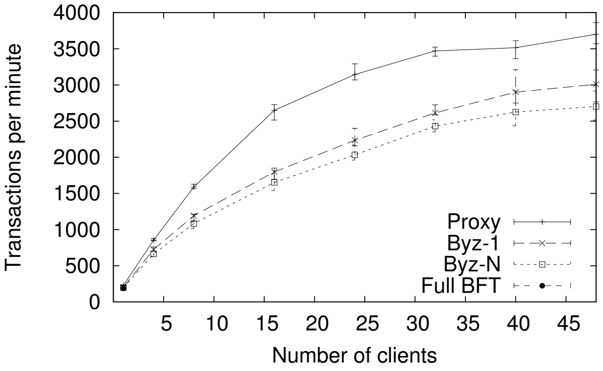

Figure 4. Performance on standard TPC-C workload (with no batches). Note that Full BFT has only a single data point, as expected, since concurrency leads to deadlock.

In our experiments, we have used the mechanism that avoids deadlocks without using database triggers. We also disabled the mechanism to verify the correctness of the execution of rolled back transactions.

7.1 TPC-C standard: read-write dominated workload

Figure 4 presents the performance results obtained with the standard TPC-C workload, consisting of 92% read-write transactions and 8% read-only transactions. The results show that the performance of our versions is between 20% to 35% lower than the *proxy* solution.

There are two main reasons for this overhead. First, the workload consists mostly of read-write transactions, for which both read and write operations must execute at all replicas. (In fail-stop replication, part of this cost is avoided as it suffices to read from a single replica.) Thus, this workload introduces overhead associated with the replication protocols that is not compensated by any form of load-balancing. This overhead could have been minimized if transactions had long prefixes of read-only operations, as explained in section 4.4. However, TPC-C read-write transactions have very small prefixes of read-only operations – e.g., the new order transaction, which may include over 50

[1] `http://hsqldb.org/`

[2] `http://sourceforge.net/projects/benchmarksql/`

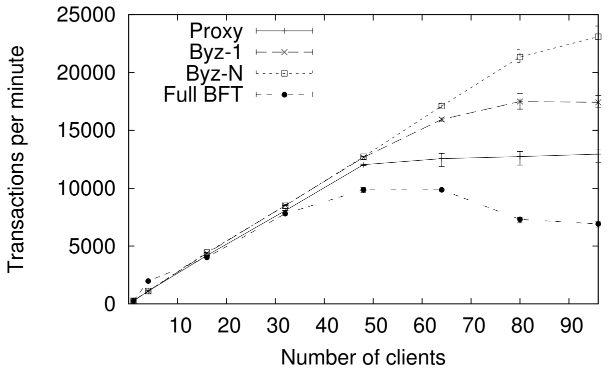

Figure 5. Performance on read-only workload, based on TPC-C transactions with no batches.

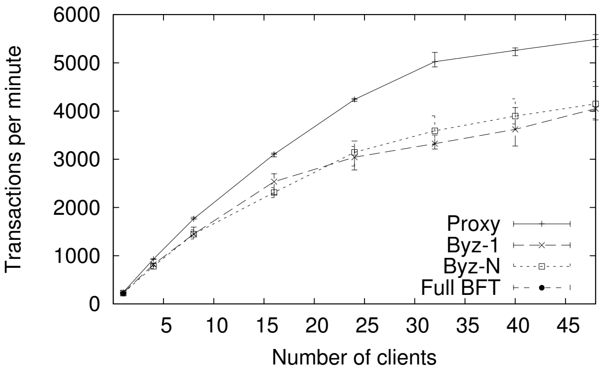

Figure 6. Performance on mixed workload, based on TPC-C transactions with no batches. Full BFT has only a single data point, as expected, since concurrency leads to deadlock.

operations, has a prefix of two read operations before the first write operation.

When compared with the *full BFT* solution, our solution performs 5% better for a single client. For a larger number of clients, the *full BFT* solution always blocked. This was expected, as we did not include any mechanism to prevent a PBFT operation from blocking in the database when trying to acquire a lock that is taken (as we have in *Byz-N*).

When comparing our two versions, the difference tends to be about 10% and slightly increasing with the number of clients, with the single master version performing better. The reason for this lies in the optimization mechanism of the single-master version, which speculatively executes operations in all replicas, thus minimizing the execution time for the commit operation.

Two additional results are important. First, the induced rollback ratio due to concurrency problems increases with the number of clients, but it is similar for both *Byzantium* versions and the *proxy* solution (with a difference of less than 3% in all studied scenarios).

The second result worth mentioning is the time to execute a transaction. In this case, the results vary depending on the transaction type. Read-only transactions run up to 1.3× faster in *Byzantium* versions than in *proxy* – we will discuss the reasons for this in more detail in the next section. Read-write transactions run slower in *Byzantium* than in *proxy* by a factor of 0.7 or better. This is due to the additional stages introduced by the replication algorithm.

7.2 Read-only workload

Figure 5 presents the performance results obtained with a modified TPC-C workload consisting of only read-only transactions, with 50% for each type of read-only transaction (check inventory level and check order status).

The results show that *Byz-N* improves on the performance of *proxy* by up to 10% when the number of clients is smaller than 32, and this improvement increases up to 90% with 96 clients. The main reason for this is related to the load of replicas. In the multi-master version, since operations of read-only transactions tend to execute only in $f + 1$ replicas, the load of each replica is half of the load when running *proxy*. Thus, by load balancing, our solution is able to achieve close to optimal throughput, almost doubling the result of *proxy*.

The performance of *Byz-1* is better than *proxy* and worse than *Byz-N*. In the case of the single master protocol, the load of replicas other than the master is at most $\frac{2}{3}$ of the load when running *proxy*.

As expected, the results show that the *full BFT* performance is the worst of the four setups, and the differences increase rapidly with the number of clients. This is due to having to use the PBFT protocol for executing each operation, and demonstrates the need for minimizing the use of this protocol, as proposed by our solution.

7.3 Mixed workload

Figure 6 presents the performance results obtained with a modified TPC-C workload consisting of 50% read transactions (with 25% for both check inventory level and check order status) and 50% write transactions (with 27% for new order, 21% for payment and 2% for delivery, keeping the original ratio among write transactions). The results show that the performance of our versions is between 20% and 30% lower than the *proxy* solution with up to 48 clients. This represents a slightly lower overhead when compared with the write-dominated workload. In this case, the results for *Byz-N* and *Byz-1* are very similar, as a result of the improved performance of the multi-master version for read-only transactions.

7.4 Byzantine behavior

Next, we evaluate the performance of Byzantium in the presence of Byzantine faults caused by database bugs. To this end, we have changed the code of the Byzantium replica proxy to simulate incorrect results from the database. The

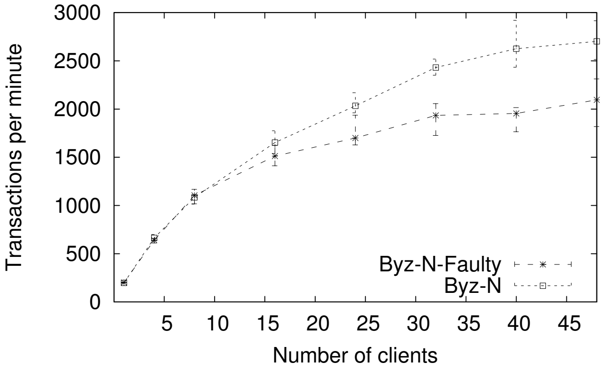

Figure 7. Performance on the presence of a single Byzantine server, using TPC-C standard workload .

implementation is very simple: for each data item read, it randomly changes the returned value with a pre-defined probability. Although this scenario represents a rather benign case of Byzantine behavior, it is one that is probably more likely to happen in practice, with a replica sending the same incorrect messages to every other node. In the scenario of a malicious replica, the performance penalty could be higher, particularly in the single-master case.

We have measured the throughput of the system with the standard TPC-C benchmark, a single incorrect replica and an error probability of 10% in the period after the server starts exhibiting Byzantine behavior. We only present the results for our multi-master design – the results with a single master show a similar pattern.

Figure 7 presents the results, with *Byz-N-Faulty* representing the throughput with a Byzantine replica. The lower throughput was expected for two reasons. First, the load is divided among a smaller number of replicas. Second, in all PBFT protocol steps, the messages from the Byzantine replica cannot be used for obtaining the required conditions, thus delaying the execution of the protocol.

A more interesting and surprising result is related to the number of rolled back transactions, which increased to about 5% of all executed transactions. This result seemed to suggest that our mechanism to change the master replica worked very slowly. However, after analyzing the experiments, we discovered that a large number of transactions with incorrect results were rolled back by the benchmark code and not due to the detection of Byzantine behavior by our algorithms. The reason for this is that the transaction code often uses the results from a previous operation as a parameter in the subsequent operations. When the previous returned result is incorrect, the subsequent operation fails and the benchmark ends up rolling back the transaction. These results also show the importance of the mechanism to verify the execution of rolled back transactions introduced in Section 4.6.

8. Conclusions

We presented Byzantium, the first proposal for middleware-based BFT replication of database systems that allows for concurrent execution of database operations and does not rely on centralized components. Byzantium shows that it is possible to obtain the strong assurances that derive from using a Byzantine fault model, while paying only a modest penalty in terms of performance overhead. We showed how to minimize the use of the expensive BFT operations using two different techniques for optimistic execution, and how to optimize the execution of read-only transactions.

We evaluated Byzantium and our results show that replication introduces only a modest performance overhead for read-write dominated workloads and we perform up to 90% better than a non-replicated database system for read-only workloads. Our single-master version performs better in read-write dominated workloads while the multi-master version performs better with a large number of read-only transactions.

In the future, we intend to deploy different database systems in different replicas, and to explore the use of other BFT protocols as their implementations become available.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Ozalp Babaoglu, for the time they dedicated to providing valuable feedback on earlier versions of this paper. This work was partially supported by CITI and FCT/MCTES project # PTDC/EIA/74325/2006.

References

[Bernstein 1986] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10715-5.

[Castro 2002] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.*, 20:398–461, November 2002.

[Cecchet 2008] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 739–752. ACM, 2008.

[Clement 2009] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright Cluster Services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290. ACM, 2009.

[DISA 2004] Defense Information Systems Agency DISA. Database security technical implementation guide - version 7, release 1. White paper available at databasesecurity.com, October 2004.

[Elnikety 2006] 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 SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 117–130. ACM, 2006.

[Elnikety 2007] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-aware Load Balancing and Update Filtering in Replicated Databases. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 399–412, New York, NY, USA, 2007. ACM.

[Elnikety 2005] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database Replication Using Generalized Snapshot Isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84. IEEE Computer Society, 2005.

[Fekete 2005] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30:492–528, June 2005.

[Fischer 1985] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[Fonseca 2010] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A Study of the Internal and External Effects of Concurrency Bugs. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2010. IEEE, July 2010.

[Garcia Molina 1986] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11:27–47, March 1986.

[Gashi 2007] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers. *IEEE Trans. Dependable Secur. Comput.*, 4:280–294, October 2007.

[Gribble 2001] Steven D. Gribble. Robustness in Complex Systems. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 21–26. IEEE Computer Society, 2001.

[Herlihy 1987] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 13–26. ACM, 1987.

[King 2005] Samuel T. King and Peter M. Chen. Backtracking Intrusions. *ACM Trans. Comput. Syst.*, 23:51–76, February 2005.

[Kotla 2007] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58. ACM, 2007.

[Lamport 1982] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[Lin 2005] Yi Lin, Kem Bettina, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 419–430. ACM, 2005.

[Nightingale 2005] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 191–205. ACM, 2005.

[Plattner 2004] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 155–174. Springer-Verlag New York, Inc., 2004.

[Preguiça 2008] Nuno Preguiça, Rodrigo Rodrigues, Cristóvão Honorato, and João Lourenço. Byzantium: Byzantine-fault-tolerant Database Replication Providing Snapshot Isolation. In *Proceedings of the Fourth conference on Hot Topics in System Dependability*, HotDep'08, pages 9–9. USENIX Association, 2008.

[Rodrigues 2001] 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*, SOSP '01, pages 15–28. ACM, 2001.

[Schneider 1990] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[Vandiver 2007] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 59–72. ACM, 2007.

[Yin 2003] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement From Execution for Byzantine Fault Tolerant Services. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 253–267. ACM, 2003.