Consistent Replication in Distributed Multi-Tier Architectures

Thomas Repantis
Akamai Technologies
Cambridge, MA 02142
trepanti@akamai.com

Arun Iyengar
IBM T.J. Watson Research Center
Hawthorne, NY 10532
aruni@us.ibm.com

Vana Kalogeraki
Dept. of Informatics, Athens University
of Economics and Business, Greece
vana@aueb.gr

Isabelle Rouvellou
IBM T.J. Watson Research Center
Hawthorne, NY 10532
rouvellou@us.ibm.com

***Abstract*—Replication is commonly used to address the scalability and availability requirements of collaborative web applications in domains such as computer supported cooperative work, social networking, e-commerce and e-banking. While providing substantial benefits, replication also introduces the overhead of maintaining data consistent among the replicated servers. In this work we study the performance of common replication approaches with various consistency guarantees and argue for the feasibility of strong consistency. We propose an efficient, distributed, strong consistency protocol and reveal experimentally that its overhead is not prohibitive. We have implemented a replication middleware that offers different consistency protocols, including our strong consistency protocol. We use the TPC-W transactional web commerce benchmark to provide a comprehensive performance comparison of the different replication approaches under a variety of workload mixes.**

***Keywords*-Replication, Consistency, Multi-Tier Architectures.**

I. Introduction

Multi-tier architectures are at the heart of modern enterprise data centers, supporting a variety of collaborative applications. Popular dynamic web applications, such as e-commerce, e-banking, social networking websites, blogs, and wikis are built using a multi-tier model. Multi-tier architectures provide separation of concerns between the presentation tier, realized by a web server, the logic tier, realized by an application server, and the data tier, realized by a database. The presentation tier accepts user requests and provides service results back to the user. The logic tier executes the required services to produce the requested results. Finally, the data tier permanently stores the state of services in a database. For example, in an online store, a sale would result to an update in the inventory. The different tiers are commonly hosted by separate machines in the same data center.

Replication is commonly employed to improve the performance [1] of such multi-tier architectures [2], [3], as well as the performance of other distributed environments, including databases [4]. In all cases, multiple replicas of the same data are maintained to tolerate failures, or concurrently serve requests and consequently balance the load among multiple machines. While replication can improve both scalability and availability, it also introduces the problem of consistently maintaining the replicated data. Two consistency models have been widely used: Strong and weak consistency. With strong consistency, all replicas of a data object served at any given time must be identical. This is not necessarily true for weak consistency, where different nodes can serve different versions of the same data at the same time. Weak consistency can cater to applications such as online forums and social networking websites. On the other hand, strong consistency may apply to e-banking, online retail stores, auction marts, and stock brokerages.

A variety of replication solutions, stemming both from academia and industry, have been proposed to address the problem of consistent and scalable data replication in multi-tier architectures [2], [3]. To increase throughput and decrease response time, all or some of the tiers are replicated among several machines, which can then participate in serving concurrent user requests. To address consistency maintenance issues, the data tier is often not replicated [5], [6]. While not replicating the data tier avoids the synchronization overhead, this non-replicated tier can become a performance bottleneck and a single point of failure. Another common approach to simplify consistency maintenance is to replicate the data tier and only offer weak consistency guarantees. For example, following a master replication scheme [7], [8], only a single master replica accepts writes, while multiple slave replicas serve reads. The updates are then propagated from the master to the slaves lazily, while the slaves continue serving stale data in the meantime. Yet another common approach for providing strong consistency when replicating the data tier, is to rely on group communication [9]. Following a group (or multi-master) replication scheme [10]–[15], multiple replicas act as masters and accept writes. A group communication layer is then used to synchronize them, so that writes are applied in the same order in all replicas. However, a group communication layer can pose scalability challenges, while its configuration involves multiple complex tradeoffs [2], [3], [9].

In this paper we provide a comprehensive study of common replication approaches that offer various consistency guarantees. By quantifying the performance overhead of providing strong consistency, we motivate a distributed strong consistency protocol that we propose. We focus on multi-tier architectures because of their popularity in real-world applications. However, the protocols we discuss may be applicable to other distributed architectures as well. We compare the performance of i) no replication, ii) partitioning, in which different nodes are responsible for different parts of the data, iii) replication

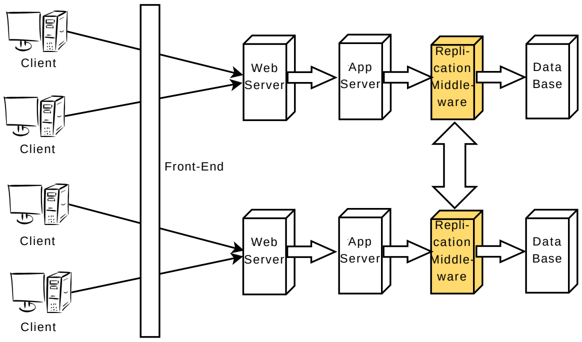


Fig. 1. Replication model.

with weak consistency, iv) replication with strong consistency using a traditional lock-based protocol, and v) replication with strong consistency using our invalidation-based, distributed protocol. Our protocol strives to minimize the communication overhead and increase performance. Unlike previous strong consistency protocols in domains such as distributed databases [16], our protocol does not require locking or lock managers. Furthermore it enables coalescing updates that have been overwritten.

We have implemented a multi-threaded Java replication middleware that offers the different consistency protocols we examine, including the strong consistency protocol we propose. The middleware lies between the logic and the data tier and is responsible for maintaining data consistent. To compare the performance of the replication approaches in real-world scenarios, we use the TPC-W transactional web commerce benchmark [17], under a variety of workload mixes. TPC-W defines various events representing user interactions with an online bookstore. We report our findings regarding the update overhead incurred, as well as the throughput and the response times attained.

The rest of this paper is organized as follows: We begin by describing our system model in section II. Section III describes the design decisions behind our strong consistency protocol, as well as its implementation. We proceed by providing an overview of different replication approaches in section IV, and comparing their design to our protocol. Section V presents the experimental comparison of all the replication approaches, while section VI discusses relevant work. Finally, section VII concludes the paper.

II. System Model

The replication model we are considering is illustrated in Figure 1. We assume a client/server architecture, in which clients use HTTP to access services provided by servers. Clients are redirected to servers via a front-end. Thus, replication is transparent to the clients. The front-end strives to distribute user requests among two or more servers using a load balancing scheme, for example round-robin. It is also session persistent. Redirection happens at the beginning of the interaction of a client with a server, and afterwards all client requests related to the same user session are directly communicating with the same server. There are several redirection solutions [18] based on application, router, or DNS enhancements, and these mechanisms are orthogonal to our design.

A server can be hosted by one or more machines, but it represents one logical unit in our model. Servers communicate with each other to maintain consistency, which is required for replication transparency. They communicate using TCP sockets and are connected using a fast network connection, for example belonging to the same LAN.

Each server assumes a tiered architecture, as shown in Figure 1. The presentation tier, implemented by a web server is responsible for interaction with the clients, while the logic tier, implemented by an application server, executes the business logic. Finally, the state of each server is stored in the data tier, implemented by a database. Our replication middleware lies between the logic and the data tiers and provides consistent replication. To maintain consistency among server replicas all accesses to the data tier are intercepted. Thus, the logic tier communicates only with our replication middleware, which is responsible for making the actual database calls. This interception can be active, by changing the application code running on the logic tier, or passive, by having the middleware provide an interface (e.g., JDBC) that makes it indistinguishable from a database from the application's perspective [15]. At each server, writes are intercepted and in addition to being applied locally they are collected and sent to the rest of the servers in the form of updates. Every server also receives updates from other servers and applies them locally. Depending on the consistency requirements, reads can also be intercepted, to ensure all updates have been applied before data are returned to the user.

Strong consistency requires that all replicas of a data object served at any point in time are identical. Thus, all updates must have been applied to all replicas before any reads can proceed. With weak consistency, on the other hand, different versions of a data object can be served at the same time, depending on the replica being accessed. In this case updates can be applied lazily, irrespective of the reads currently taking place.

The consistency protocols we will be discussing apply to any kind of data objects. However, in the application environment of multi-tier architectures we are studying in this work, the objects we are interested in maintaining consistent represent tables in the relational database of the data tier.

The metrics signifying the system's performance, which we are interested in optimizing, are throughput and response time. Throughput is measured in user interactions per second sustained by the system. Response time is the end-to-end time spent to serve a client request, and includes the processing time to execute the application logic and access the database, as well as the communication time to transfer the request and the reply between the client and a server.

III. Invalidation-Based Strong Consistency

In this section we describe our invalidation-based, distributed, strong consistency protocol, and its implementation in our replication middleware. The next section reviews other common consistency protocols and compares them to ours.

A. Replication Middleware

Our distributed strong consistency protocol is implemented in a multi-threaded Java replication middleware. The middleware is responsible for maintaining the data tiers of all server nodes consistent. We describe now how the copies of all tables in the database of each node are kept consistent, assuming all nodes maintain copies of all tables. We discuss dividing tables across nodes in section IV-B. Each node follows a three-phase protocol when it needs to update its copy of a data table. The protocol ensures that the copies of the table held by other nodes will be updated as well, and that all nodes will be serving the same version of each table at all times. The protocol has three phases for updating a table, which include the exchange of corresponding messages: The *Invalidation Request*, the *Invalidation Reply*, and the actual *Update*. Invalidation requests and replies are used to coordinate the updates on the shared data. Detailed descriptions of the protocol and its different phases are given in sections III-B and III-C respectively.

Figure 2 shows the building blocks of our replication middleware, responsible for consistency maintenance of the local tables and for communication with the remote nodes of the distributed architecture. There are six major modules. The *Replica Access* module is responsible for providing permission to access tables in the local node and for sending Invalidation Requests and Updates to the remote nodes. The *Invalidation Requests Manager* module keeps track of the Invalidation Requests the local node makes and notifies the replica access module once all remote nodes have replied to an Invalidation Request. The *Invalidations Manager* module keeps track of the tables that have been invalidated and notifies the replica access module once an invalid table has been updated with its new version. The *Message Sender* module is responsible for sending Invalidation Requests, Invalidation Replies, and Updates to the remote nodes. The *Message Receiver* module is a thread listening for incoming messages. Once a message is received, a new thread that implements the *Message Processor* module is spawned to apply an Update, or to determine whether an Invalidation Request should be positively or negatively acknowledged. For efficiency, we can avoid the creation and destruction of threads, by maintaining a pool of them equal to the number of tables in the database.

When a node s1 needs to read a table t1 in the database, the Replica Access grants that permission once the Invalidations Manager notifies it that table t1 is not invalid as currently being updated remotely. If t1 is currently being updated, the notification takes place once the update is completed. We use thread communication to implement these notification mechanisms. Concurrent reads from multiple nodes on their local copies of the same table, which has not been invalidated, proceed independently. Thus, concurrent reads are allowed and do not require communication with remote nodes.

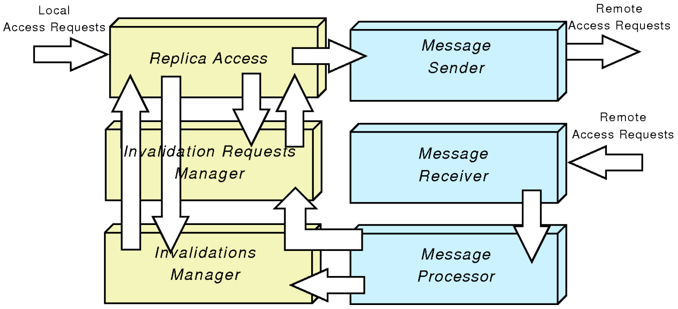


Fig. 2. Replication middleware implementation modules.

B. Distributed Protocol Operation

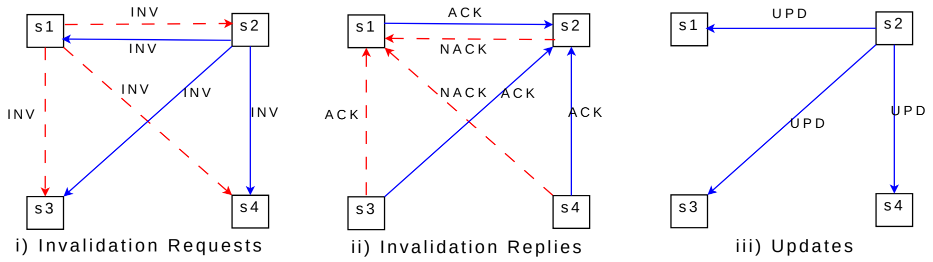
We now describe our strong consistency protocol, before laying out the details of its different phases in the following section. Figure 3 shows an example of a protocol execution, while algorithms 1 and 2 outline the execution on the sender and receiver side of an update respectively.

When a node s1 needs to update a data table t1, it sends invalidation requests to the rest of the nodes that have copies of the same table. After s1 has received positive invalidation replies from *all* the nodes, indicating that they have invalidated their copies, it can update t1.

When a node s2 receives s1's invalidation request to invalidate its copy of table t1, it responds with a positive or a negative invalidation reply, depending on whether it will do so or not. To decide whether it will invalidate its copy of t1, s2 checks whether it has sent or received an invalidation request for t1. If that has been the case, s2 compares the timestamp of that invalidation request to the timestamp of the invalidation request of s1. The request with the earlier timestamp is rejected. If both timestamps are equal, an ordering of the nodes is imposed, for example by comparing the hashes of each nodes' IP and port and having the request from the node with the largest hash value dominate. If the invalidation request of s1 was the one with the latest timestamp, s2 invalidates t1 and sends a positive invalidation reply to s1. If the invalidation request of s1 was the one with the earliest timestamp, s2 sends a negative invalidation reply to s1.

Node s2 sends a negative invalidation reply if it has sent or received an invalidation request with a later timestamp for the same table t1. If s2 has sent an invalidation request for t1, it is about to update it and its update will overwrite s1's update since it has a later timestamp. If s2 has received an invalidation request for t1, another node is about to update it and its update will be the one to overwrite s1's update, again because it has a later timestamp.

Node s1 updates table t1 after it has received positive invalidation replies from all other nodes maintaining copies of t1. If s1 receives one or more negative invalidation replies, it aborts its attempt to update t1 and its update will be ignored. When s1 will receive the invalidation request for t1 from



i) Invalidation Requests ii) Invalidation Replies iii) Updates

Fig. 3. Sample message exchange between 4 nodes. In dashed lines the messages pertaining to s1's update attempt. In solid lines the messages pertaining to s2's update attempt. INV: invalidation request, ACK: positive invalidation reply, NACK: negative invalidation reply, UPD: update.

s2 (with a timestamp which was later than that of its own invalidation request) it will invalidate its copy of t1 and send s2 a positive invalidation reply. Consequently, after s2 will receive positive invalidation replies that all other nodes including s1 have invalidated their copies of t1, it will proceed with the update.

It is important to note that the protocol correctness only depends on the replies of the nodes that are actually intending to perform an update. This is because they will *always* send a negative invalidation reply if they have a later timestamp, whereas other nodes may or may not do so, depending on the order with which these other nodes receive the invalidation requests. Figure 3 shows an example of this case. In this example both s1 and s2 send invalidation request messages. s2's timestamp dominates and therefore its update is applied. Notice that even though s3 acknowledges positively to s1 because it received its request before the request of s2, the protocol execution is successful. This is because s2 will always send a negative invalidation reply to s1, since s2 has a later timestamp. Therefore, s1 will never proceed with its update, since that would require a positive invalidation reply from all other nodes, including s2. In other words, the protocol correctness only depends on the replies of the nodes that are actually intending to perform an update (in this case s2's response to s1 (and s1's response to s2)). These replies do not depend on the ordered delivery of messages since they only require local information. In the example of Figure 3, s2 only requires its local timestamp to decide sending a negative invalidation reply to s1. Since s2's invalidation reply will be negative, and since s1 would require positive invalidation replies from all nodes to proceed, s1 will not perform its update. This is true regardless of the replies of the nodes that are not intending to perform an update (s3 and s4) (which depend on the order with which these nodes received s1's and s2's invalidation requests). This explains why the protocol does not require an atomic broadcast primitive to order the delivery of messages.

C. Protocol Phases

We now describe in detail the three different protocol phases, Invalidation Requests, Invalidation Replies, and Updates, as well as their implementation in the replication middleware which we illustrated in figure 2.

1) Invalidation Request: User requests received by nodes result to updates in the data stored in the tables. When a node s1 needs to write a table t1, similarly to when reading a table, access needs to be granted. The Replica Access grants that permission once the Invalidations Manager notifies it that table t1 is not invalid as currently being updated remotely.

After the Invalidations Manager in node s1 has ensured that t1 is not currently being updated, s1 sends Invalidation Requests to all remote nodes that host copies of the table. The Invalidation requests contain the name of the table that is requested to be invalidated and a timestamp of when the request was generated. The timestamp reflects the local time at node s1 that initiates the request. While the clocks of all nodes can be loosely synchronized, the correctness of the protocol does not depend on that. As explained in the next paragraphs, the timestamps are used for the nodes to agree on a relative order between them, and could therefore be arbitrary. Finally, s1 also calls the Invalidation Requests Manager, which notifies it once all remote nodes have replied.

2) Invalidation Reply: The remote nodes send Invalidation Replies to node s1 that requested an invalidation. Invalidation Replies specify the table the Invalidation Request referred to and whether the Invalidation Request is granted (positive reply) or not (negative reply). We now describe how nodes decide whether they will send a positive or a negative Invalidation Reply.

When a node s2 receives an Invalidation Request from node s1 to invalidate table t1, the Message Processor consults the Invalidation Requests Manager to determine whether s2 has already sent Invalidation Requests for t1. If this is the case, the timestamps of the local (s2's) and the remote (s1's) Invalidation Requests are compared. The request with the later timestamp dominates and the one with the earlier timestamp will need to be retried. The Invalidations Manager on each node keeps track of pending invalidation requests by that local node and signals when they can be retried.

If both timestamps are equal, an ordering of the nodes is imposed, as was described in section III-B. If the local (s2's) Invalidation Request dominated, a negative Invalidation Reply is sent from s2 to s1. If the remote (s1's) Invalidation Request dominated, the Invalidations Request Manager is called to register the fact that the local request is unsuccessful, and a positive Invalidation Reply is sent from s2 to s1. The

Invalidations Manager at s2 is called to register the fact that table t1 is being updated.

When a node s3, that has not sent Invalidation Requests for t1, but has received an Invalidation Request for t1 from s2, receives an Invalidation Request for t1 from s1, it performs the timestamp comparison described above and sends a positive or negative Invalidation Reply to s1. If s1's Invalidation Request dominates, s3 calls its Invalidations Manager to register the fact that t1 is now being updated by s1 instead of s2.

3) Update: If at least one of the Invalidation Replies was negative, this signifies that an update with a later timestamp will be applied on t1 by another node. Therefore s1 postpones its attempt to write on t1, notifying the Invalidation Requests Manager that the request was unsuccessful and needs to be retried. If all Invalidation Replies were positive, the update can proceed. The Invalidation Requests Manager is called to register the fact that the request was successful and the Invalidations Manager is called to register the fact that table t1 is currently being updated. The table is updated locally and then Update messages are sent to the remote nodes.

Update messages are used to prefetch the table changes to the remote nodes. They contain the name of the table that is to be updated, the SQL statement that needs to be executed on that table, and the parameters of that statement. By executing the same statements with the same parameters on the same tables, remote nodes maintain their data consistent.

After the Updates are sent, the Invalidations Manager at s1 is called again to register the fact that t1 is not being updated anymore. The Invalidations Manager at s2 is called again when the Update message for t1, sent from s1, is received by s2, to register the fact that t1 is not being updated anymore. The Invalidations Manager on each node awakes any pending invalidation requests for t1, which had been unsuccessful and can now be retried. In more fine-grained implementations, for example when applying updates on a tuple- instead of a table-level, if multiple nodes attempt to concurrently update the same tuple, one writer can simply overwrite the other. In that case, updates can simply be coalesced, by applying only the latest one and ignoring the rest.

Algorithm 1 Sender algorithm.

```
SubmitUpdateRequest
Input:   Local invalidation request $INV_t$ for table $t$
         Number of replicas $S$
Output:  False (cancel request) or
         True (grant request and send update $UPD_t$)
for each replica $s_i$ in $S$
    send invalidation request $INV_t$ to $s_i$
invalidation replies = 0
while invalidation replies $< S$
    receive invalidation reply
    invalidation replies++
    if invalidation reply is negative ($NACK_t$)
        return False
return True
```

Algorithm 2 Receiver algorithm.

```
ApproveUpdateRequest
Input:   Remote invalidation request $INV_t$ for table $t$
Output:  Positive invalidation reply $ACK_t$ or
         Negative invalidation reply $NACK_t$
if (exists local or remote invalidation request $INV_t$' and
    timestamp($INV_t$') $>$ timestamp($INV_t$))
    return $NACK_t$
else
    return $ACK_t$
```

IV. Replication Approaches Background

Having proposed our invalidation-based strong consistency protocol, we now compare it to common replication approaches and their corresponding consistency protocols.

A. Weak Consistency

In a weak consistency protocol different nodes can serve different versions of the same data at the same time [7], [8]. When a node needs to read a table in the database, no communication with other nodes is required. Consequently, concurrent reads are allowed.

When a node needs to write a table in the database, it does so locally and then sends update messages to all other nodes, all of which maintain copies of that table. Nodes continue serving content before receiving and applying updates. Therefore different versions of the same table may be served by different nodes at the same time.

For applications that do not require an accurate global state, weak consistency can provide good performance, since both read and write requests are served locally and thus don't have to block. Examples of applications for which users can tolerate seeing a slightly different state of the system include online forums and social networking websites. For instance, users add responses to a conversation much slower than the speed with which changes are propagated among nodes. Similarly, it is acceptable that different users may see a change on somebody's profile with a small time variation.

B. Partitioning

A common technique to increase database performance is to partition the data among several machines, which is supported by all major DBMSs. A protocol that maintains data consistent by partitioning them across nodes provides strong consistency but no replication. In such a protocol data are distributed across nodes. Partitioning of the data can happen in different ways: i) Different tables can be stored in different nodes. ii) A table can be divided horizontally among multiple nodes, with its rows distributed across nodes, or iii) A table can be divided vertically among multiple nodes, with its columns separated into different tables and them stored in different nodes. Common partitioning criteria include whether the partitioning key of the table falls within a range of values, whether it is included in a list of values, whether its hash has a certain value, or a combination of the above.

A request to read or write a particular tuple in a table is redirected to the node responsible for storing it. No communication with other nodes is required for either reading or writing a tuple, since a single copy of each tuple exists in the system. Thus, a node hosting the single copy of a data tuple serves all reads and writes for it.

The advantage of partitioning is that it provides strong consistency, without requiring communication between nodes for reads or writes. The disadvantage is that no concurrent read or write requests for the same tuple can be served, as only a single copy of each tuple exists in the system. Still, performance is improved in comparison to having a single node serving the requests for all tuples, since requests for different tuples can be served by different nodes simultaneously. However, queries that touch multiple tables may have to contact multiple nodes before they can be executed.

Partitioning is a good approach for applications that do not have a lot of concurrent requests for the same data tuple, as those have to be served sequentially by the single node that stores it. Partitioning, similarly to the next consistency protocol we describe, provides strong consistency, and is therefore suitable for a variety of applications that require it, such as e-banking, online retail stores, auction marts, or stock brokerages. Availability however may suffer, since the data are not replicated across nodes.

In case partitioning is combined with replication and multiple but not all nodes maintain copies of each data table, the invalidation-based strong consistency protocol we described in section III can still be applied. The difference in this case would be that every time a table needs to be updated, instead of all the nodes, only the nodes that maintain copies of that particular table would need to be contacted.

C. Lock-Based Strong Consistency

Locks are a common approach for ensuring strong consistency [16], [19]. A lock-based strong consistency protocol can be implemented either in a centralized or in a distributed way. In the centralized implementation there is a single node acting as the lock manager, responsible for acknowledging requests for reading or writing a table. In the distributed implementation all nodes with copies of a table have to acknowledge requests for reading or writing it. The centralized implementation has the disadvantage of rendering one node a single point of failure and a potential performance bottleneck. However, it may reduce the number of message exchanges. We describe the centralized implementation, and our description applies to the distributed implementation as well, with the only change being the one mentioned above.

When a node needs to read a table, it sends a readlock acquire message to the lock manager. The readlock is granted once the lock manager sends back a readlock grant message. The node releases the readlock once it has read the table, by sending a readlock release message to the lock manager. If two readlock acquire messages are sent concurrently by two nodes, both acquire the readlock.

Concurrent reads are allowed, i.e., a node that needs to read a table that is readlocked can do so. The node sends a readlock acquire message to the lock manager, which replies with a readlock grant message. The node releases the readlock after it has read the table, by sending a readlock release message to the lock manager.

When a node needs to write a table, it sends a writelock acquire message to the lock manager. The lock manager replies with a writelock grant message. If the table is readlocked, the lock manager sends the writelock grant message only after it has received a readlock release message from all nodes that were reading the table. Thus, a node has to wait for the readlock release before being able to write a table.

A node releases the writelock once it has written the table, by sending a writelock release message to the lock manager. In the distributed implementation of the protocol, the update is usually piggybacked on the writelock release message to minimize the communication overhead. In the centralized implementation of the protocol, besides the writelock release, additional update messages may be required for the update to reach all nodes.

If two or more writelock acquire messages for the same table are sent concurrently by two nodes, a tiebreaker is used in the distributed implementation of the protocol, whereas in the centralized implementation the lock manager serializes concurrent requests. An example of a tiebreaker is a predefined order of the nodes, used to prioritize their requests.

Nodes that need to read or write a table that is writelocked wait for the writelock release. They send readlock or writelock acquire messages to the lock manager. The lock manager replies with the corresponding grant message only after it has received a writelock release message from the node that was writing the table.

In comparison to a lock-based strong consistency protocol, the invalidation-based strong consistency protocol we described in section III avoids the use of locking and lock managers. While similarly to two-phase locking [16], [19], invalidation-based strong consistency divides the operation in phases, it does so only for write and not for read operations. Thus, in comparison to lock-based strong consistency, invalidation-based strong consistency may increase the number of requests served concurrently, as read requests for tables that have not changed do not require any synchronization between nodes. In contrast, a lock-based strong consistency protocol requires the acquisition of a readlock.

V. Experimental Evaluation

A. Experimental Setup

We used the TPC-W transactional web commerce benchmark [17] to compare the performance of the different replication approaches, including our invalidation-based strong consistency protocol. TPC-W is an industry-standard, e-commerce

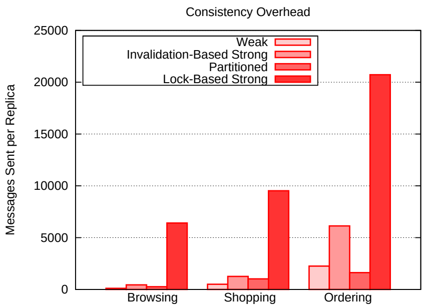


Fig. 4. Communication overhead (sent messages per replica) to maintain consistency.

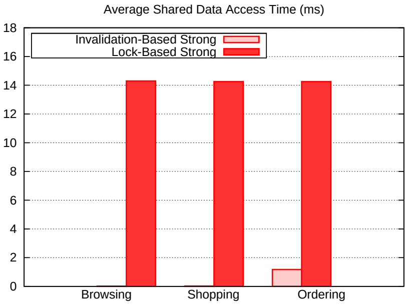


Fig. 5. Average time (ms) to access a shared data table.

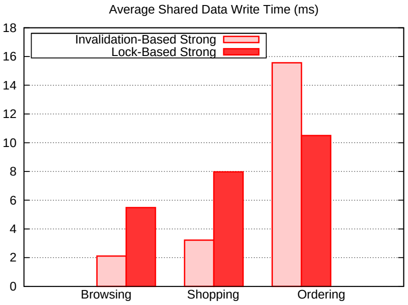


Fig. 6. Average time (ms) to write to a shared data table.

benchmark that emulates an online bookstore. It defines the operation of the store, as well as the workload. The benchmark emulates multiple concurrent HTTP sessions, browsing and buying products from the website. The workload includes contention on data access and data update. The operation is based on dynamic page generation with database accesses, and the database consists of tables with various sizes, attributes, and relationships. We used the Java implementation of TPC-W by the University of Wisconsin-Madison [20].

We instantiated four servers, each one consisting of an application server, a DBMS, and our replication middleware. We also experimented with higher replication degrees in section V-D. Requests were generated from a separate host, based on the benchmark workload, and were redirected randomly among the different servers. We used Apache Tomcat 6.0.13 as our application server and MySQL 5.0.41 as our DBMS. The implementation of the replication middleware consists of approximately 3000 lines of Java code. It includes all the replication approaches discussed in section IV, as well as our invalidation-based strong consistency protocol presented in section III. In partitioning we divided among servers tables that were being updated, to save on consistency-related communication, without affecting application correctness. No updates were exchanged for these tables, as each server was the only one holding a copy of the corresponding table partition.

TPC-W specifies three different workload mixes: *Browsing* consists of 95% browsing interactions, such as displaying information about products, and 5% ordering interactions, such as adding items to a shopping cart. *Shopping* consists of 80% browsing and 20% ordering. Finally, *ordering* consists of 50% browsing and 50% ordering. We conducted experiments with a duration of 1000 seconds, and an additional 100 seconds of ramp up and 100 seconds of ramp down times. The requests were choosing among 144000 customers and 10000 items.

The primary metrics of TPC-W are WIPS and WIRT. WIPS refers to the average number of completed Web Interactions Per Second and measures the system throughput. WIRT refers to the average Web Interaction Response Time and measures the end-to-end time elapsed before a client request receives a response.

B. *Consistency Overhead*

In the first set of experiments we compared the overhead of the various consistency protocols with regards to communication and data access.

Communication overhead. Figure 4 shows the messages sent per replica to maintain consistency. We show the results for the different workload mixes. We observe that as the number of write operations increases in the workload, the number of messages exchanged increases as well. The lock-based strong consistency protocol incurs a higher number of messages than our invalidation-based strong consistency protocol, since messages need to be exchanged for reading in addition to writing data. We have implemented the distributed version of the lock-based protocol. A centralized implementation would reduce the messages exchanged, but possibly overload a particular node. Partitioning the data reduces the communication overhead significantly.

Access time. Figure 5 shows the average time required to access a shared data table, in ms. Weak consistency and partitioning are not displayed, since they do not require coordination with remote nodes and are therefore much faster. Since lock-based strong consistency requires message exchanges for both reads and writes we observe a high shared data access time, regardless of the workload. Our invalidation-based strong consistency protocol only pays the communication price when writing data, which is more common in the ordering workload. We see that invalidation-based strong consistency significantly reduces the average access time compared to lock-based strong consistency, especially for mostly read-only workloads.

Write time. Figure 6 shows the average time required to write a shared data table, in ms. As the number of writes increases in the workload, there is a higher possibility that writes will coincide in time. In this case invalidation-based strong consistency can incur additional overhead to serialize them. Even though the average time to write on shared data is higher for invalidation-based strong consistency, that price is paid less frequently than with lock-based strong consistency. This is because writes to shared data are much more frequent in the latter case, as is shown in Figure 8. Again, weak consistency and partitioning are not displayed, since they do not require coordination with remote nodes and are therefore much faster.

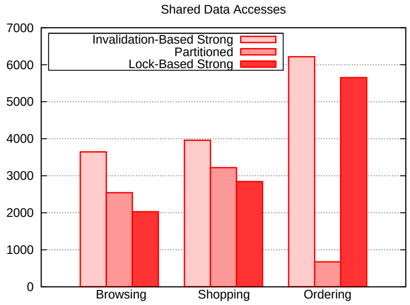


Fig. 7. Number of accesses of shared data.

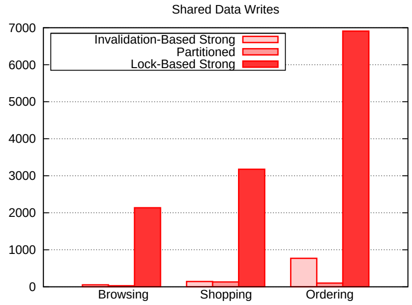


Fig. 8. Number of writes to shared data.

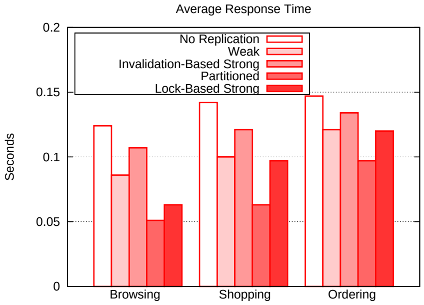


Fig. 9. Average response time (s) for different consistency protocols and workloads.

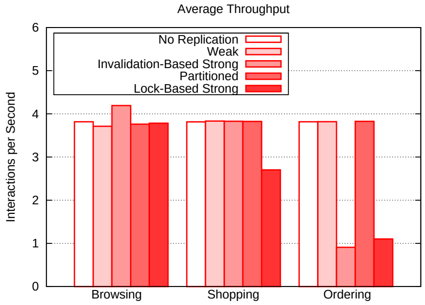


Fig. 10. Average throughput for different consistency protocols and workloads.

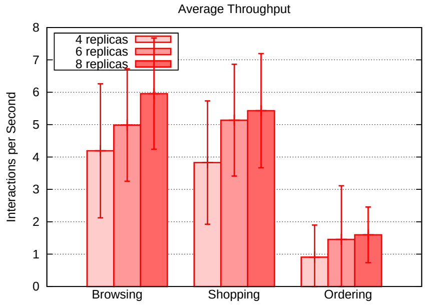


Fig. 11. Average throughput for increasing replicas (invalidation-based strong consistency).

Data accesses. Figure 7 shows the number of accesses to shared data, which also depends on the throughput sustained by the different protocols. We observe that partitioning decreases the amount of shared data accesses by orders of magnitude in the case of the ordering workload. This shows that carefully choosing to partition tables that are being updated we can eliminate a lot of the shared data accesses and consequently the communication overhead they incur. Weak consistency is not shown, since none of the data accesses are shared, in the sense that they do not require online node coordination.

Data writes. Figure 8 shows the number of writes to shared data. As expected, the number of writes increases as the workload becomes more write-heavy. Again, partitioning eliminates a lot of shared data writes, while lock-based strong consistency requires significantly more shared data writes. Weak consistency is not shown, for the same reason as in the previous figure.

C. Replication Performance

In the second set of experiments we compared the performance of the various replication schemes, focusing on the TPC-W metrics of response time (WIRT) and throughput (WIPS). We also compared the performance of the replication schemes to the performance of a single server serving user requests ("No Replication").

Response time. Figure 9 shows the average response time in seconds, for the different consistency protocols, as well as for the case of using a single server. We observe that maintaining strong consistency has significant overhead, which however still does not outweigh the replication benefit. Strong consistency is not significantly more expensive than weak consistency. Partitioning on the other hand provides significant benefits in terms of response time, by decreasing the communication overhead between nodes. It is important to analyze response time in conjunction with throughput, as serving less user requests can decrease response time. This is for example the case with lock-based strong consistency.

Throughput. Figure 10 shows the average throughput, in completed interactions per second, again for the different replication schemes. We observe that the write-heavy ordering workload decreases the throughput of the strong consistency protocols. This is because of the required synchronization between nodes. The latter is avoided by partitioning. The throughput during read-heavy workloads, on the other hand, is not affected by our invalidation-based strong consistency protocol. The throughput of the lock-based strong consistency protocol however decreases even during the shopping workload, due to the use of readlocks. Thus readlocks result in throughput decrease even for primarily read-heavy workloads.

D. Invalidation-Based Strong Consistency Scalability

In the third set of experiments we focused on the scalability of our invalidation-based strong consistency protocol.

Replication degree. Figure 11 shows how the average throughput is affected as more replicas are added to the system. We observe that read-heavy workloads can see a larger performance benefit from increased replication. This is because less updates need to be coordinated with remote replicas. Additionally, the performance benefit of replication decreases as the replication degree increases. This can be explained by the fact that larger numbers of replicas require more messages to be exchanged in order to maintain consistency, especially for write-heavy workloads. Replication degrees such as the ones shown in Figure 11 are common for online applications up to a medium scale. Related work has also used similar setups, ranging from 1 to 9 servers [5]–[8], [13], [15], [21].

VI. Related Work

Consistency tradeoffs have been explored when replicating for availability. The CAP dilemma states that a replication system can provide only two out of sequential Consistency, high Availability, and resilience to network Partitions [22], [23]. Reduced consistency can be traded off with increased availability [24], [25]. TACT [25] provides a continuous consistency model to bound the provided availability as a function of numerical error, staleness, or order error.

The performance overheads of consistency have first been studied in the domain of distributed databases. [4] has presented an average case analysis for major replication models. The taxonomy presented includes eager replication (strong consistency) versus lazy replication (weak consistency), as well as group replication, in which all replicas accept writes versus master replication, in which only a master replica accepts writes.

Solutions to the consistency problem in the domain of distributed databases can be divided in three categories: Lock-based, timestamp-based, and optimistic approaches. In all cases each replica is responsible for serializing the transactions that access local tables, and the additional protocol each time is used to enforce global serialization. Lock-based approaches [16] use a lock manager to achieve this goal. Timestamp-based approaches [16] utilize the generation of an agreed timestamp ordering between the replicas. Finally, optimistic approaches [26], [27] require global validation before a transaction is committed, or a global ordering on committing transactions.

[2], [3] provide overviews of replication approaches in order to enhance web application performance of multi-tier architectures. Having a single data tier supporting multiple web and application servers is a common replication scheme [5], [6]. While not replicating the data tier avoids the synchronization overhead, this non-replicated tier can become a performance bottleneck and a single point of failure.

Master (or master-slave) replication, in which reads and writes are separated, is another common replication model [7], [8]. In this model only a single master replica accepts writes, while multiple slave replicas serve reads. Having a single master solves the synchronization and consistency maintenance issues. To improve performance however, updates are only propagated asynchronously from the master to the slave replicas. Thus, this model offers only weak consistency.

In group (or multi-master) replication, in which multiple replicas act as masters and accept writes, a group communication system is used to synchronize them, so that writes are applied in the same order in all replicas. This is the approach employed in several systems [10]–[15]. Write conflicts may lead to retries, presenting potentially a scalability bottleneck [4]. Moreover, group communication, in addition to requiring significant message exchanges, also calls for balancing complex implementation and configuration tradeoffs [2].

In distributed versioning [21], [28] and snapshot isolation [7], [12] multiple versions of data tables are maintained, so that concurrency can be increased. In [21], [28] multiple nodes operate concurrently on data and reads are redirected by a centralized, conflict-aware scheduler to a node with a current copy of the table they require. Having a centralized scheduler offers tight consistency control but can potentially limit scalability. In [12] multi-version caching is employed at the application server in addition to the database. In this case a multicast-based group communication system is needed to synchronize the replicas.

Gossiping has also been proposed as a means to replicate the soft state of the logic tier in multi-tier architectures [29], in order to minimize accesses to the data tier. This technique however only offers weak consistency.

Data partitioning, as was explained in sections IV-B is another technique that aims at increasing concurrency in group replication, allowing each node to be responsible for a subset (partition) of the data. In the case of database replication, partitioning takes place either horizontally (dividing rows across nodes), or vertically (dividing columns). Common partitioning criteria include a range, a list of values, a hash function, or a combination of the above, applied to the partitioning key of a table. All major DBMSs support partitioning, including IBM DB2, Oracle, Microsoft SQL Server, PostgreSQL, and MySQL. While partitioning provides performance benefits such as the ones outlined in section V, the data are not replicated, which may have availability implications. Additionally, queries that touch multiple tables may need to travel through more than one nodes.

[30] discusses replication for TPC-W, which is also the application benchmark we use for our performance comparison. The authors discuss improvements in availability and performance that can be obtained by taking into account application-specific semantics. Focusing on the particular application, the authors present a design for distributed objects that each manages a specific subset of shared information. While we also use TPC-W for our evaluation, the invalidation-based strong consistency protocol we propose makes no application-specific assumptions.

VII. Conclusion

In this paper we have presented common replication approaches with various consistency guarantees, and proposed an efficient, distributed, strong consistency protocol. We have compared the design of our protocol to established approaches and discussed its implementation in a replication middleware for multi-tier architectures. We have used the TPC-W transactional web commerce benchmark to conduct a comprehensive experimental evaluation of five different replication approaches. In our experiments we have compared the performance of no replication to that of partitioning, as well as to the performance of replication with weak consistency, with lock-based strong consistency and with our invalidation-based strong consistency. We have provided an experimental comparison using a variety of realistic workloads to demonstrate the practicality of our invalidation-based strong consistency protocol. Our implementation of different common replication approaches allowed us to *quantify* the performance differences among them, and in particular to quantify the performance hit of providing strong consistency depending on the workload type.

REFERENCES

[1] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel, "Performance impacts of autocorrelated flows in multi-tiered systems," *Performance Evaluation*, vol. 64, no. 9–12, pp. 1082–1101, October 2007.

[2] E. Cecchet, A. Ailamaki, and G. Candea, "Middleware-based database replication: The gaps between theory and practice," in *Proceedings of ACM SIGMOD*, June 2008.

[3] S. Sivasubramanian, G. Pierre, M. Steen, and G. Alonso, "Analysis of caching and replication strategies for web applications," *IEEE Internet Computing*, vol. 11, no. 1, pp. 60–66, January/February 2007.

[4] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proceedings of ACM SIGMOD*, June 1996.

[5] L. Cao and M. T. Ozsu, "Evaluation of strong consistency web caching techniques," *World Wide Web*, vol. 5, no. 2, pp. 95–124, 2002.

[6] M. H. S. Attar and M. T. Ozsu, "Alternative architectures and protocols for providing strong consistency in dynamic web applications," *World Wide Web*, vol. 9, no. 3, pp. 215–251, October 2006.

[7] C. Plattner and G. Alonso, "Ganymed: Scalable replication for transactional web applications," in *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, October 2004.

[8] C. Plattner, G. Alonso, and M. T. Ozsu, "DBFarm: A scalable cluster for multiple databases," in *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, November 2006.

[9] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 427–469, December 2001.

[10] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, 2000.

[11] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso, "Middle-R: Consistent database replication at the middleware level," *ACM Transactions on Computers*, vol. 23, no. 4, pp. 375–423, 2005.

[12] F. Perez-Sorrosal, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Consistent and scalable cache replication for multi-tier J2EE applications," in *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, November 2007.

[13] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Enhancing edge computing with database replication," in *Proceedings of 26th Symposium on Reliable Distributed Systems (SRDS)*, September 2007.

[14] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu, "On the performance of consistent wide-area database replication," in *Johns Hopkins University, Center for Networking and Distributed Systems (CNDS) Technical Report CNDS-2003-3*, December 2003.

[15] E. Cecchet, M. Julie, and W. Zwaenepoel, "C-JDBC: Flexible database clustering middleware," in *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[16] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, vol. 13, no. 2, pp. 185–221, June 1981.

[17] Transaction Processing Performance Council, "TPC Benchmark W (Web Commerce) Specification," February 2002.

[18] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal, "Active names: Flexible location and transport of wide-area resources," October 1999.

[19] G. Ricart and A. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Communications of the ACM*, vol. 24, no. 1, pp. 9–17, January 1981.

[20] T. Bezenek, T. Cain, R. Dickson, T. Heil, M. Martin, C. McCurdy, R. Rajwar, E. Weglarz, C. Zilles, and M. Lipasti, "Characterizing a Java implementation of TPC-W," in *Proceedings of the 3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads (CAECW)*, January 2000.

[21] K. Manassiev and C. Amza, "Scaling and continuous availability in database server clusters through multiversion replication," in *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2007.

[22] E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, July/August 2001.

[23] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, pp. 51–59, June 2002.

[24] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[25] H. Yu and A. Vahdat, "The costs and limits of availability for replicated services," *ACM Transactions on Computer Systems*, vol. 24, no. 1, pp. 70–113, February 2006.

[26] H. Kung and J. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, June 1981.

[27] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, "Efficient optimistic concurrency control using loosely synchronized clocks," in *Proceedings of ACM SIGMOD*, May 1995.

[28] C. Amza, A. Cox, and W. Zwaenepoel, "Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites," in *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, June 2003.

[29] T. Marian, M. Balakrishnan, K. Birman, and R. Renesse, "Tempest: Soft state replication in the service tier," in *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[30] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, "Improving availability and performance with application-specific data replication," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 106–120, January 2005.