Enhancing Edge Computing with Database Replication

Yi Lin* Bettina Kemme* Marta Patiño-Martínez+ Ricardo Jiménez-Peris+

*McGill University, School of Computer Science, Canada
+Facultad de Informatica, Universidad Politecnica de Madrid, Spain
E-mail: (ylin30,kemme)@cs.mcgill.ca, (mpatino,rjimenez)@fi.upm.es

Abstract

As the use of the Internet continues to grow explosively, edge computing has emerged as an important technique for delivering Web content over the Internet. Edge computing moves data and computation closer to end-users for fast local access and better load distribution. Current approaches use caching, which does not work well with highly dynamic data. In this paper, we propose a different approach to enhance edge computing. Our approach lies in a wide area data replication protocol that enables the delivery of dynamic content with full consistency guarantees and with all the benefits of edge computing, such as low latency and high scalability. What is more, the proposed solution is fully transparent to the applications that are brought to the edge. Our extensive evaluations in a real wide area network using TPC-W show promising results.

1 Introduction

Edge computing has emerged as an important technique for delivering Web content over the Internet to a constantly growing user base. Edge computing has its roots in content delivery networks (CDNs) that deliver content by moving it from centralized servers to the edge of the network, closer to end-users. This reduces communication over the wide-area network (WAN) and the load on the server since some of the computation can be performed at the edge. Edge computing is evolving from caching static content web pages to supporting more sophisticated applications with dynamic content. Specifications for edge computing have been developed, such as Edge Side Includes (ESI) which enables to distinguish between cacheable and non-cacheable content, and provides facilities to aggregate and assemble content at the edge. However, these techniques are neither generic nor transparent, requiring customization on a per-application basis. Moreover, caching is difficult if the data can be updated frequently. In this case, the cached data at the edge might become stale compared to the current state at the server. While this might be acceptable for some applications, for an important class of applications (e.g., payment operations) transactional semantics are a must and updates are frequent.

We propose a radically different approach for update intensive edge computing applications. These applications have at their core databases. By moving a copy of the database to the edge and keeping the copies coordinated, adapting the application becomes unneccessary. We present a wide area replication protocol that enables the delivery of dynamic content with full consistency guarantees and with all the benefits of edge computing, such as lower latency and higher scalability. At the same time, the solution is transparent to the application. The proposed technique exploits a novel consistency criterion, 1-copy-snapshot isolation (1-copy-SI) [14], that enables high scalability and contrasts with former approaches for database replication based on 1-copy-serializability (1CS) [11] that offered poor scalability and were not amenable to be used in WANs [13].

Our approach differs from lazy replication approaches that are typically used for wide area replication to overcome the lack of scalability of 1CS. In lazy replication, transactions are executed first at one replica. Any updates are propagated to other replicas only after transaction commit, thus providing fast response times. However, late propagation can lead to serious inconsistencies, e.g., selling the same item concurrently at two different replicas. To simplify the problem, primary-secondary approaches have been proposed. They require all updates to be performed at a primary replica. The secondary copies can only execute read-only transactions. Secondaries receive a serialized stream of updates propagated from the primary that guarantees consistent albeit possibly stale reads. However, transparency is lost, since the system must know a priori whether a transaction is read-only or not. Furthermore, performance is affected because response time for update transactions might include a WAN message round for each operation.

Many replication solutions have been proposed that provide full data consistency by coordinating transaction execution before commit (known as eager replication) [1, 3, 4,

16, 17, 14]. However, their focus was on replication within a data center and not to deliver content to the edge of the network, i.e., they were designed for local-area networks where message latency is not an issue. Hence, some of them use several messages per transaction. Some of them [2, 3, 16] also suffer from transparency problems.

In this paper, we propose a replication tool specifically designed for WAN replication to deliver dynamic content transparently to the edge of the network. Our approach is based on a middleware architecture that separates replication issues from standard database tasks. The system has the potential to support a heterogeneous set of database systems, e.g., supporting both Oracle and PostgreSQL database systems. The design of our replication middleware was driven by the aim of overcoming the drawbacks of existing approaches described before. Firstly, replication is transparent to the application such that the application does not have to be modified to enhance it with edge computing. Secondly, there is at most one WAN message round within the response time of update transactions. Thirdly, data is consistent at all times so the delivered dynamic content is always fully consistent. The proposed approach has been benchmarked with TPC-W, the benchmark for web servers with dynamic content in a real wide area network with different latencies between main and edge servers and a heterogeneous setup.

2 Execution Model

2.1 Transaction execution model

User requests that require database access are always encapsulated in transactions. User requests that only access static content are considered non-transactional. A transaction (typically implemented as an application program) contains a sequence of read $r(x)$ and write $w(x)$ operations on the data objects in the database, and ends with a *commit/abort (c/a)* operation. Read-only transactions only have reads while update transactions contain at least one write operation. A transaction represents a logical unit of work (in terms of business logic) and should be executed atomically, i.e., either all or none of its operations are successfully processed. Furthermore, concurrent conflicting transactions need to be isolated from each other via a concurrency control method.

Snapshot Isolation (SI) is a common transaction isolation level (e.g., used by Oracle, PostgreSQL, Microsoft SQL Server). Although it does not provide *serializability* as traditionally defined in textbooks, it provides the ANSI serializable isolation level. Using SI a transaction sees a committed snapshot of the database as of start of transaction, and only write/write conflicts are detected. Each update on a data object x creates a new version of x that becomes visible only when the updating transaction commits. A transaction T reading x sees the last committed version of x as of start time of T. Thus, in contrast to *serializability*, reads and writes do not conflict and never block each other. Furthermore, if a transaction T wants to update x and there is a transaction T' that also updated x and committed after T started (i.e., it is concurrent to T), then T must abort.

2.2 Edge Server Architectures

We study five different architectures for providing dynamic content edge services. In all approaches, clients that are local to the main server connect to it directly and their execution is always local. In the centralized approach (Fig. 1), all requests go to a central server, independently of the location of the users. Remote users always experience long response times. Typical existing edge server systems follow one of the two cases depicted in Fig. 2. Clients connect to their local servers. In case (i) of Fig. 2 the web-server forwards all requests to the local application logic where they are executed. If a request is transactional, the application logic makes calls to the remote enterprise database to perform the database operations (resulting in one WAN message round for each database operation). In case (ii) of Fig. 2, the web-server is able to distinguish between transactional and non-transactional requests. It redirects non-transactional requests to the local application logic while transactional requests are forwarded to the application logic at the main server that then executes the transaction and accesses the database locally. In this case, one WAN round trip message is needed for forwarding the request and returning the result.

Fig. 3 shows edge services based on lazy primary replication. Note that Fig. 3 is different from Fig. 2 in that there is a database copy at each edge server. Now, additionally to non-transactional requests also read-only transactions will be served locally at the local cache or local database copy. Database operations of update transactions must be executed at the primary database. For that, the edge server must know whether a request triggers an update transaction or is read-only. In Fig. 3 case (i) the local application server executes update transactions leading to a WAN message round for each database access (i.e., many rounds per transaction). In Fig. 3 case (ii) the web-server sends the request for an update transaction directly to the application server at the primary server, leading to one WAN message round per transaction. We name cases (i) and (ii) as **LPn-Msg** (Lazy Primary with n Messages) and **LP1Msg** respectively for our future discussion. For both approaches, the primary database propagates any updates to the secondary databases on a regular basis or immediately after commit. This does not affect the client response time but the data at the edge is stale until changes are propagated. This has two

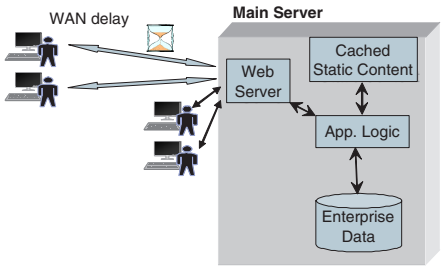

Figure 1. Centralized system

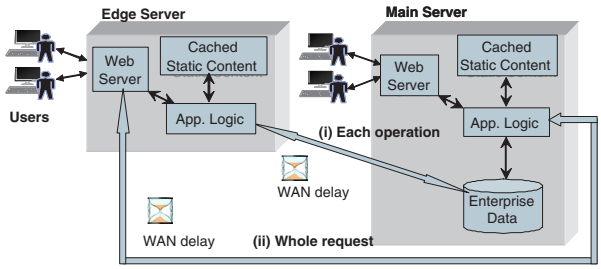

Figure 2. Edge-servers sharing a centralized DB

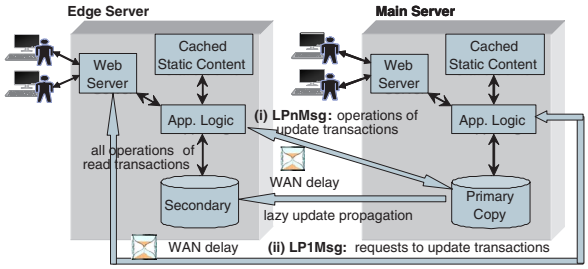

Figure 3. Edge servers based on lazy primary

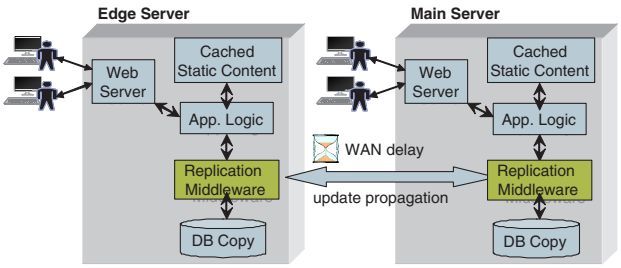

Figure 4. Edge servers with our approach

negative effects. Firstly, a client might not see its own updates if it first submits an update transaction (which is executed at the primary database) and then a read-only transaction (which reads the local, possibly stale data copy). Secondly, crashes might lead to lost data, if the main server crashes before propagating some changes to the secondary which takes over as primary.

3 SEQ: replica control for WAN

We aim at a solution where the message overhead is kept as small as possible and transaction execution can be distributed evenly among all servers. Furthermore, there is no restriction on client applications and clients should always see current and consistent data. Our architecture is depicted in Fig. 4. There is a replication middleware instance for each DB copy at each sever. All requests are processed locally. Neither web- nor application server need to perform redirection or be aware of distribution and replication. All database access is to the local copy. Only for update transactions the middleware instances coordinate leading to one WAN message round per transaction.

Our replication infrastructure provides 1-copy-SI [14] which requires the replicated database to behave as if there was one logical copy of the database that provides snapshot isolation (SI). As discussed in Section 2.1, SI is more attractive than serializability in that read and write operations do not conflict and never block each other. Only writes are relevant for conflict detection. This makes SI attractive for replication since determining read sets is difficult for middleware based approaches. Furthermore, SI is particularly attractive for a WAN setting since no information about reads needs to be exchanged among the replicas.

3.1 Protocol Overview

Read-only transactions are executed locally and committed immediately without any synchronization. An update transaction is first executed at the local replica. During execution, the middleware keeps track of all modified objects identified by their primary key in form of a writeset. There is no need to keep track of read operations because they are not needed for conflict detection. When the commit request is submitted, the writeset is sent to the middleware at the main server. We refer to this middleware as the global sequencer. The global sequencer, upon receiving a writeset from transaction T from replica R, performs a validation that detects whether it had received conflicting writesets from concurrent transactions at other replicas. In this case, it informs R that T must abort. Otherwise, it informs R that T can commit. Furthermore, it also sends the writeset to all other replicas. Writesets are multicast in validation order and remote replicas apply the writesets in the order they receive them from the sequencer.

In this scheme, the WAN communication overhead is one message round for update transactions executed at edge servers. Otherwise, no WAN communication is needed. Note that the replication middleware has already explicit

information of a transaction upon its commit time. This nicely eliminates the need of predefining the transactions (e.g., read-only or not). Furthermore, read operations of update transactions are executed at their local edge servers instead of the main server. Thus, SEQ has better load balancing potentials than primary replication.

3.2 Protocol details

The details of our protocol, denoted as SEQ (for sequencer) are depicted in Figure 5. The underlying database replicas are assumed to have a concurrency control mechanism that provides SI[1]. The application logic of the servers uses a standard database interface, submitting a transaction operation by operation to its local middleware replica M^k (step 2). When a transaction begins (2a), it is assigned a timestamp, $start$, referring to the last committed transaction. This will later allow the identification of all concurrent transactions. All read and write requests are simply forwarded to the local database replica R^k (2b). When the application logic submits the commit request (2c), M^k extracts the writeset from R^k. If a transaction is read-only, it commits immediately. Otherwise, M^k sends the writeset to the sequencer middleware replica M^{SEQ} for validation.

Once M^{SEQ} receives the writeset for transaction T (3), it validates T. Validation must run in a critical section. If validation succeeds (3b), T is assigned a validation timestamp (increasing counter) and is stored for validation of future transactions. Validation succeeds, if none of the concurrent, already validated transactions (those whose validation timestamps are higher than T's start timestamp), conflicts with T[2]. M^{SEQ} then sends the commit decision including the writeset and the validation timestamp to all middleware replicas (including itself) in one FIFO message. If the validation fails (3c), M^{SEQ} sends an abort notification back to the local replica of T.

Once a middleware replica M^k receives a commit message for transaction T (4), it appends T to the $tocommit\_queue$. If M^k receives an abort notification for T (5), it asks R^k to abort T. A transaction T can be applied at M^k (only needed if not a local transaction) and commit once all previously delivered transactions have been committed (6). Then M^k sets $lastcommitted\_vid$ to be T's validation timestamp. Since transactions are committed in validation order, this is the same as increasing $lastcommitted\_vid$. Note that starting and committing transactions at the database are serialized with a mutex in order to guarantee that the correct $lastcommitted\_vid$ is assigned to the $start$ value of newly starting transactions.

[1]Note that some commercial database systems use a special locking mechanism to handle write/write conflicts early in the execution of a transaction. In this case, deadlocks might occur as discussed in [14]. In this paper, we ignore this issue. Our protocol can be easily adjusted to such situation similar to the approach taken in [14].

[2]A garbage collection process is responsible for deleting a writeset once there are no transactions in the system that are concurrent to the corresponding transaction.

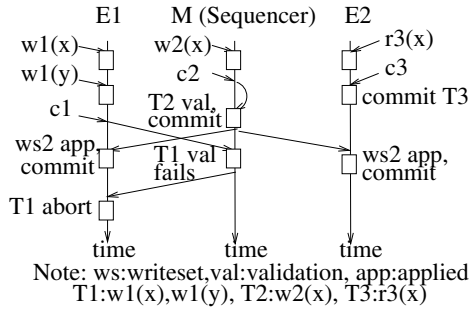

Figure 6. Example execution with SEQ

Due to space limitations, we do not show the proof of correctness here. Please refer to [15] for detail. We rather show an example for the better understanding of our protocol.

Example 1 *Fig. 6 shows an example with three transactions T_1:$w_1(x)$, $w_1(y)$, T_2:$w_2(x)$ and T_3:$r_3(x)$ running concurrently at sites $E1$, M and $E2$, respectively. M's middleware replica is the sequencer. Transactions execute at their local database first. T_3 reads a snapshot of the database not including any changes of concurrent transactions T_1 and T_2 (as it would be in a centralized database). After execution, T_3 can commit immediately since it is read-only. T_1's writeset is sent to the sequencer for validation. For T_2 no communication is needed since it executed at the sequencer site. Assume the sequencer at M validates first T_2 and then T_1. T_2's validation succeeds because there is no transaction validated so far. The sequencer sends T_2's writeset to all sites where it is executed and committed. T_2 is also committed at M. The validation of T_1 fails since concurrent conflicting transaction T_2 was allowed to commit. The sequencer simply sends the abort decision back to T_1's local middleware at $E1$ where it is aborted. That is, all three sites commit T_2 and abort T_1.*

4 HYBRID

While optimized on performance, SEQ has the same fault tolerance shortcoming as lazy primary copy replication. If the main server acting as sequencer crashes, some transactions committed at the main server but not propagated yet might be lost. Replication protocols based on *Group Communication Systems* (GCS) avoid this problem by using *uniform reliable* multicast for inter-replica communication [1, 17, 16, 19, 14, 23]. Uniform reliable multicast guarantees that once a replica receives a message all

1. Initialization:
   - $next\_vid := 1$ (M^{SEQ} only)
   - $ws\_list := \{\}$ (M^{SEQ} only)
   - $tocommit\_queue := \{\}$
   - $lastcommitted\_vid := 0$
   - $dbmutex, wsmutex$
2. Upon receiving an operation Op of T
   (a) if Op is the first operation of T
   - obtain $dbmutex$
   - $T.start := lastcommitted\_vid$
   - begin T at the local R^k
   - release $dbmutex$
   - execute in local R^k and return to application logic

   (b) else if Op is read or write, then
   - execute in local R^k and return to application logic

   (c) else (commit)
   i. T.WS:=getwriteset(T) from local R^k
   ii. If T.WS=$\emptyset$ commit and return to application logic
   iii. Send T to M^{SEQ}
3. M^{SEQ} only: Upon receiving T from M^j
   (a) obtain $wsmutex$
   (b) if !$\exists$ T' $\in ws\_list$ such that T.start<T'.vid $\wedge$ T.WS $\cap$ T'.WS $\neq \emptyset$:
   - $T.vid := next\_vid++$
   - append T to ws_list
   - send (COMMIT, T) to all middleware replicas in FIFO order
   - release $wsmutex$

   (c) else
   - release $wsmutex$
   - send (ABORT, T) back to M^j
4. Upon receiving (COMMIT, T) from M^{SEQ}
   - append T to $tocommit\_queue$
5. Upon receiving (ABORT, T) from M^{SEQ}
   - abort T and return to the application logic
6. Upon T is first in $tocommit\_queue$
   - if T is remote begin T at R^k and apply $T.WS$
   - obtain $dbmutex$
   - commit T at R^k
   - $lastcommitted\_vid := T.vid$
   - release $dbmutex$
   - remove T from $tocommit\_queue$
   - if local, return to application logic

Figure 5. SEQ protocol on middleware replica M^k

replicas receive the message unless they crash. With this, it is impossible that a replica receives a transaction (or the decision for a transaction), commits the transaction, but nobody else knows about the transaction. These protocols are also different from SEQ in that they do not have a sequencer. Instead, a replica multicasts all relevant information to all other replicas using *total order* multicast, which guarantees that all replicas receive all messages in the same order. Now each replica uses that order to detect conflicts and decide on the commit/abort of transactions. Since all receive the messages in the same order, all replicas make the same decisions. However, while these protocols work well in LAN settings, they are unfeasible for WAN environments. [13, 20] have shown that total order and uniform reliable multicast has prohibitively long message delays in WANs already with two or three sites.

However, we can still take advantage of GCS in some configurations. In large configurations, the main server, and even some edge servers, might have several database replicas (as they also have web-server and application server farms). For example, a Chinese news website might have its main server with many replicas in the company's headquarter located in Beijing, an edge server with several replicas in Shanghai, and then other edge servers with only one replica scattered around the world. For these kinds of applications, we propose the HYBRID approach, which addresses the fault tolerance and performance issues. An example of its architecture is depicted in Fig. 7. Replicas are split into different subsets, each of them being located on a different LAN. We assume the main server to have at least two replicas.

Among the replicas within the main server, we use a replica control protocol which is based on GCS and provides 1-copy-SI, e.g., SRCA-REP in [14]. Since communication is fast in a LAN, the overhead of uniform reliable, total order delivery is acceptable. For the secondary LANs, we use hierarchical validation. A transaction is first validated by a local sequencer at the edge server. If validation succeeds, the local sequencer forwards it to a replica at the main server for further validation. Decisions are sent from the primary LAN via the local sequencers to other replicas.

HYBRID improves over SEQ in several ways. First, since the replicas at the main server use uniform reliable delivery, no transaction will be lost unless all replicas crash. Secondly, at the edge servers only the local sequencers perform WAN communication, and only these local sequencers must be known by the replicas at the main server. This also leads to less WAN messages since commit decisions are not sent to all remote replicas but only to the local sequencers within each edge cluster which forward them in their local LANs. Moreover, only the sequencer in a LAN will have an open port on the firewall for WAN access. It reduces the chances for attacks and the complexity of network management. Finally, part of the validation is done at the local sequencers, decreasing the validation load at the main server.

1. Initialization:
   - $next\_vid := 1$ (replicas at main server)
   - ws_list:={} (replicas at main server and $M^{localSEQ}$)
   - $max\_vid := 1$ ($M^{localSEQ}$ only)
   - $tocommit\_queue := \{\}$
   - $lastcommitted\_vid := 0$
   - $dbmutex, wsmutex$
2. If M^k at edge server:
   - See steps 2,4 5, and 6 of Figure 5
3. If M^k=$M^{localSEQ}$ at edge server (besides step 2):
   (a) Upon receiving T from M^j in the same server
      i. obtain $wsmutex$
      ii. if !$\exists T' \in ws\_list$ such that T.start<T'.vid $\wedge$ T.WS $\cap$ T'.WS $\neq \emptyset$:
         - $T.start := max\_vid$
         - release $wsmutex$
         - send T to $M^{globalSEQ}$
      iii. else
         - release $wsmutex$
         - send $(ABORT, T)$ back to M^j
   (b) Upon receiving (COMMIT, T) from $M^{globalSEQ}$
      - obtain $wsmutex$
      - $max\_vid := T.vid$
      - append T to ws_list
      - send (COMMIT, T) to all M^j in the same LAN in FIFO order.
      - release $wsmutex$
   (c) Upon receiving (ABORT, T) from $M^{globalSEQ}$
      - send (ABORT,T) to the originator of T
4. If M^k in main server:
   (a) Upon receiving an operation Op of T
      - Same as Figure 5 step 2 except step 2ciii: Multicast T locally in uniform reliable and total order.
   (b) Upon receiving T sent by a $M^{localSEQ}$ from an edge server ($M^{globalSEQ}$ only)
      - Multicast T locally in uniform reliable and total order.
   (c) Upon receiving T multicast in total order
      i. if !$\exists T' \in ws\_list$ such that T.start<T'.vid $\wedge$ T.WS $\cap$ T'.WS $\neq \emptyset$:
         - $T.vid := next\_vid++$
         - append T to ws_list
         - append T to $tocommit\_queue$
         - if M^k is $M^{globalSEQ}$, send (COMMIT,T) to all $M^{localSEQ}$ in FIFO order
      ii. else
         - if T local, abort T and return
         - else if T originated at edge server and M^k is $M^{globalSEQ}$, send (ABORT, T) back to the $M^{localSEQ}$ of the originator of T.
   (d) Upon T is first in $tocommit\_queue$
      - See step 6 of Figure 5

Figure 8. HYBRID protocol on middleware replica M^k

4.1 Protocol details

For the sake of simplicity we assume a single replica at the main server to communicate with all local sequencers. We refer to this replica as global sequencer (note, however, that validation is done at all replicas at the main server). The algorithm can be easily extended such that each replica at the main server maintains the communication with some of the local sequencers.

We show the details of the protocol in Fig. 8. When a transaction is submitted to a replica in an edge server, it follows the same procedure as discussed in SEQ (Step 2) until it passes the validation in the sequencer of this edge server (3(a)ii). At this time, it can not commit yet because there may be some concurrent conflicting transactions at other servers. Hence, its writeset has to be sent to the global sequencer for *global validation*. However, its $start$ value is adjusted so that validation is not repeated for those transactions for which already the local validation confirmed that there is no conflict. When the global sequencer receives a transaction from an edge server (4b), it multicasts the writeset in uniform reliable and total order among the replicas at the main server. When a transaction is submitted to a replica at the main server (4a), it follows the same procedure as discussed in SEQ protocol except that its writeset will be multicast in uniform reliable and total order locally. Thus, all transactions are delivered to all replicas at the main server (4c). They validate transactions according to the delivery order. Thus, all decide in the same way. If a transaction passes validation it is enqueued for execution. Moreover, the global sequencer sends in FIFO order the commit decision and the writeset to all the local sequencers which forward it to the others replicas of their LANs (3b). Thus all replicas will execute and commit the transaction. If validation fails (4(c)ii) and it was a transaction of the main server, the corresponding replica aborts the transaction. Otherwise, the global sequencer notifies the local sequencer of the originator of the transaction about the abort. This local sequencer forwards this decision to the originator (3c). Replicas at the main server apply writesets as in SEQ (4d).

5 Performance evaluation

5.1 System description

In our evaluation, we compare SEQ and HYBRID against lazy primary copy replication (LP1Msg and LPnMsg). In order to provide a fair comparison, all approaches were implemented within a single Java based middleware replication framework. As application, we use the TPC-

W standard industrial benchmark for e-commerce applications. TPC-W models an Amazon-like e-bookseller. It is a good example of a web-based application with a substantial amount of dynamic content. We chose the shopping mix with has 20% write transactions in order to show the performance of both read-only and update transactions. We use a standard setup of 100,000 items and 100 emulated browsers which leads to a database with 650MB. The database fits into memory and all our experiments are CPU-bound. All our experiments use PostgreSQL7.2 as database. For HYBRID we used Spread [22] as group communication system.

We conducted our experiments in a WAN with 1-4 sites at Montreal (Canada), 1-3 sites at Madrid (Spain), 2 sites at Toronto (Canada), and 1 site at Edmonton (Canada). All machines are PCs with similar computing power (e.g., AMD 1.5-3.0GHZ/0.5-2GB memory/Linux). The round trip times between machines vary from 40 to 150 ms depending on the distances. To achieve the desired system-wide load, clients were evenly distributed among the main and edge servers and submitted transactions at the same rate. In all tests, aborts occurred. However, aborts in the worst case never exceeded three percent (for most of the experiments they were close to zero), and are not further discussed. All tests achieved the confidence interval of 95% $\pm$2.5%.

5.2 Scenario 1: Individual Edge Servers

In this first scenario we compare SEQ against the two lazy primary copy approaches using 4 servers in 4 different cities. We show the results for the main server (at Montreal) and at the edge server (Madrid) that has the longest network distance from the main server.

We first analyze the CPU usage at the different servers since it has a quite large effect on the response time of the different algorithms. Fig 9.(a) shows the CPU usage (cpu graphs) at the main server, i.e., the primary server for LP1Msg and LPnMsg, and the sequencer for SEQ. SEQ has a significant lower CPU usage than the lazy primary copy approaches, especially at high loads. With primary copy, both read and write operations of all update transactions are executed at the primary. In contrast, with SEQ, the read operations of update transactions submitted to edge servers are processed only at the edge servers, keeping the load at the primary lower. LPnMsg has slightly higher CPU usage than LP1Msg because LPnMsg has to process more messages than LP1Msg. At the edge servers (Fig 9.(d)), SEQ has higher CPU usage than lazy primary copy for exactly the same reason that it distributes the load more evenly across the servers.

We now look at the response times of read-only transactions submitted to either the main or the edge servers ($total$ graphs in Fig. 9.(a) and (d)), and the time spent within the database (DB bars). We can observe that these times are directly correlated with the CPU usage because read-only transactions do not have any communication overhead. Thus, SEQ has lower response than lazy primary copy at the main server (Fig. 9.(a)) and higher response time at the edge servers (Fig. 9.(d)). Furthermore, most of this response time is due to time spent in the database.

Let us now examine the behaviour of update transactions in Fig 9.(b) and (e). The figures show the response time of update transactions ($total$ graph), the time spent at the database (DB bars) and at the network ($network$ bars). Update transactions submitted to the main server (Fig. 9.(b)) are mainly affected by the time spent at the DB since there is no WAN communication. The DB time is directly correlated with the CPU usage. Thus, since the SEQ has the lowest CPU usage, it provides the shortest response times. LPnMsg and LP1Msg have similar response times since they have similar CPU usage. Update transactions submitted to the edge servers (Fig. 9.(e)) show a different picture. Note that the y-axis scales to 1000 ms compared to 250 ms for the other figures. The response time of LPnMsg is four times higher than for LP1Msg and SEQ. The reason is that LPnMsg needs one WAN message round per operation (and in TPC-W an update transaction has on average four operations) while SEQ and LP1Msg only need one per transaction. The network bars in the figure show that LPnMsg clearly has higher communication overhead than LP1Msg and SEQ. The figure also shows that both LP1Msg and LPnMsg have higher DB overhead than SEQ. This is because the update operations are executed at the primary database. We have seen before that the primary server in the lazy primary approaches has a higher CPU usage than the edge servers with SEQ, leading to longer execution times. Therefore, also LP1Msg has larger response times than SEQ at the edge servers.

We have also evaluated the bandwidth consumption at the main server and edge servers (Fig. 9.(c) and (f)) since bandwidth usage is another crucial factor that edge computing aims at reducing. At the main server (Fig. 9.(c)) LPnMsg has the highest incoming and outgoing bandwidth consumption because of the large number of messages needed. LP1Msg has higher bandwidth consumption than SEQ because the main server must return query results of update transactions in LP1Msg but not in SEQ. The edge server (Fig. 9.(f)) has similar tendency as the main server.

5.3 Scenario 2: Clustered Edge Servers

In this scenario we compare HYBRID against LP1Msg (being the better of the lazy primary protocols) and SEQ in the network topology shown in Fig 7. Recall that HYBRID provides a higher level of fault-tolerance than SEQ

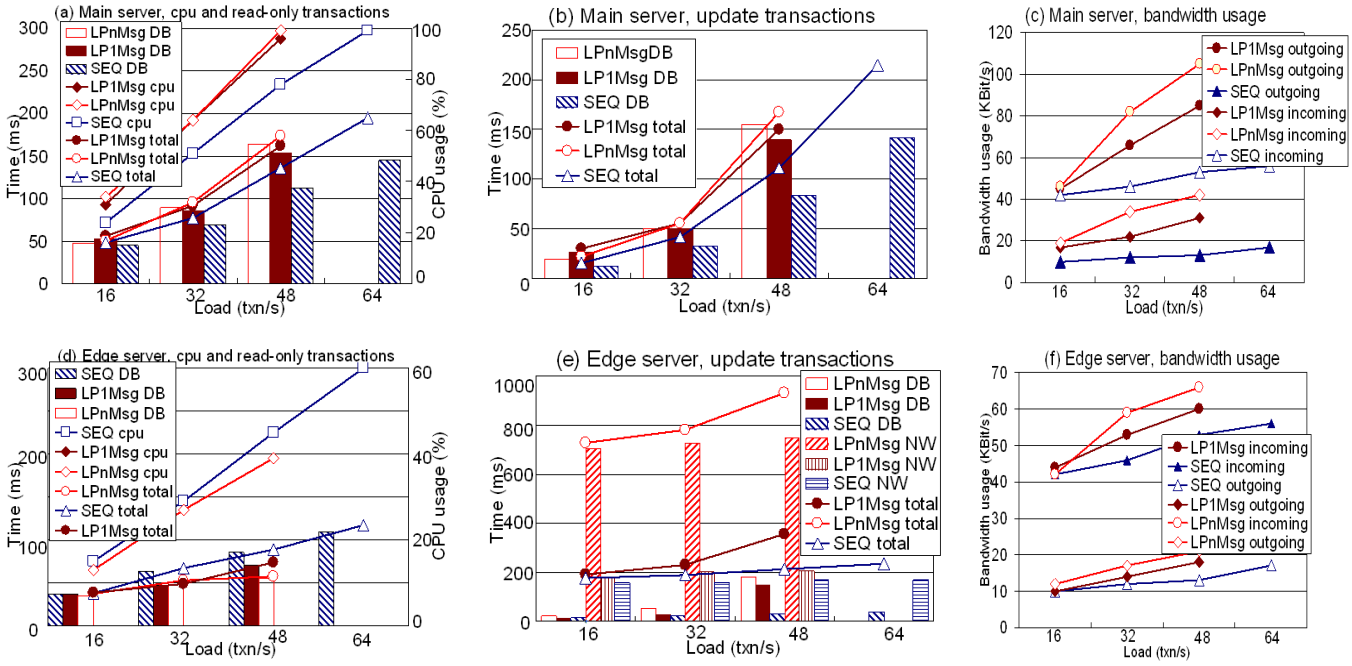

Figure 9. Scenario 1: individual servers (Main server + 3 Edge servers) at four cities

and LP1Msg. We study the results (1) at the global sequencer (primary), (2) at other replicas in the primary LAN, (3) at a local sequencer and (4) at other replicas in the secondary LANs.

Fig. 10.(a)-(d) show the response time (*total* graphs) and the time spent in the DB (*DB* bars) for read-only transactions submitted to different servers, and the CPU usage (*CPU* graphs) at these servers. DB overhead is the main overhead of read-only transactions. LP1Msg has the highest CPU usage, and thus, the largest response time (*total* graphs) for read-only transactions at the primary in the primary LAN (main cluster) (Fig. 10.(a)). At all other replicas (Fig. 10.(c)-(e)), it has slightly lower CPU usage than HYBRID and SEQ, and thus slightly lower response times. Comparing HYBRID with SEQ, HYBRID has slightly higher CPU usage due to the overhead of the GCS, but response times remain similar for read-only transactions.

Figure 10.(e)-(h) show the behavior of update transactions. The figures contain graphs for the response time (*total*), the time at the DB and at the network. Independently to which server an update transaction is submitted, LP1Msg has the worst response times except for transactions submitted to the primary server at very low loads. The reason is that all operations of update transaction must be executed at the primary database. Comparing HYBRID with SEQ, both spend similar time in the DB. However, HYBRID has the additional cost of the GCS resulting in higher response times. However, the difference is relatively small (20-50 milliseconds or 15% in most cases). This is the cost of stronger fault tolerance provided by HYBRID.

6 Related work

Many replica control protocols in commercial solutions and research are lazy primary [9, 6, 18]. We have already discussed that lazy-primary is not transparent to applications since they need to know a priori whether transactions are read-only or update. Our experiments show that lazy primary has worse performance than SEQ and even HYBRID with its stronger fault-tolerance properties. Other approaches, although allowing update transactions to be executed everywhere, use a central scheduler to synchronize each operation [4, 3]. Thus, similar to LPnMsg, they are not feasible for a WAN setting. Note that although SEQ also uses a centralized scheduler, it only synchronizes once per transaction.

In recent years many replica control protocols have been proposed [1, 16, 17, 7, 19], taking advantage of total order and uniform reliable multicast primitives provided by GCS [5] for synchronization before commit. [7], in fact, provides generalized snapshot isolation which is similar to 1-copy-SI. [13] showed that protocols based on GCS do not perform well in WANs. Some protocols [2, 16] require all operations of a transaction to be known at start time or to multicast the readset that can be very large. The former is

not transparent due to the need for application information, and the latter would result in large bandwidth consumption and large latencies in a WAN.

There exist some proposals directly addressing dynamic content for edge computing. Traditional approaches [24] cache static content and access the database at the main server to assemble dynamic pages for database data. ESI [8] is a standard for this technique. However, this technique lacks transparency, and is implemented on a per application basis. [10] proposes per-object replication strategies such as primary copy for some types of objects and reconciliation mechanisms for other types of objects. Thus, transparency is lost and the replication architecture is highly interwoven with the application semantics. The authors of [12] provide edge server primary-copy replication of J2EE objects (beans) with transactional properties. However, the solution is specific for application servers based on J2EE. Finally, [21] proposes a lazy primary-copy DB approach where different edge servers might have different partitions of the database. For each partition, a different edge server might be primary. Our approach differs in two aspects. Firstly, we provide full consistency, 1-copy-snapshot isolation, while [21] uses lazy propagation. Furthermore, we can execute all transactions locally, without complex redirection, achieving a very good load balance across servers.

7 Conclusions

In this paper we have presented a novel approach to delivering dynamic content in edge computing. Unlike current approaches either based in static content caching or primary-copy replication, our approach provides a fully consistent update-everywhere solution where all transactions are always executed at the local replica. Update transactions only incur one WAN message round per transaction, and hence, their performance is similar to the fastest existing replication solutions. Different to nearly all existing solutions, our approach is transparent to the application. The proposed approach results in higher scalability and lower latencies for delivering dynamic content. And finally, our protocols provide a high degree of fault-tolerance, especially HYBRID, and are able to adjust to the network topology of the application.

References

[1] Y. Amir, C. Danilov, et al. On the Performance of Consistent Wide-Area Database Replication, 2003.

[2] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of ICDCS*, 2002.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End DBs of Dynamic Content Web Sites. In *Middleware*, 2003.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Conference*, 2004.

[5] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4), 2001.

[6] K. Daudjee and K. Salem. Lazy Database Replication with Ordering Guarantees. In *ICDE*, 2004.

[7] S. Elnikety, F. Pedone, and W. Zwaenepoel. DB Replication Using Generalized Snapshot Isolation. In *SRDS*, 2005.

[8] Edge Side Includes (ESI). http://www.esi.org/.

[9] S. Gancarski, H. Naacke, E. Pacitti, and P. Valduriez. Parallel Processing with Autonomous DBs in a Cluster System. In *CoopIS*, 2002.

[10] L. Gao, M. Dahlin, et al. Application specific data replication for edge services. In *ACM WWW*, 2003.

[11] J. Gray, P. Helland, P. O'Neil, et al. The Dangers of Replication and a Solution. In *SIGMOD*, 1996.

[12] A. Leff and J.T. Rayfield. Alternative edge-server architectures for enterprise javabeans applications. In *Middleware*, 2004.

[13] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Consistent Data Replication: Is it feasible in WANs? In *Euro-Par*, 2005.

[14] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *SIGMOD*, 2005.

[15] Yi Lin. *Practical and Consistent Database Replication*. PhD thesis, School of Computer Science, McGill University, to be submitted in 2007.

[16] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Consistent Database Replication at the Middleware Level. *ACM Trans. on Comput. Syst. (TOCS)*, 23(4), 2005.

[17] F. Pedone, R. Guerraoui, and A. Schiper. The DB State Machine Approach. *Distributed and Parallel Databases*, 2003.

[18] C. Plattner, G. Alonso, and M. T. Ozsu. Dbfarm: A scalable cluster for multiple dbs. In *Middleware*, 2006.

[19] L. Rodrigues, H. Miranda, R. Almeida, et al. Strong Replication in the GlobData Middleware. In *Works. on Dependable Middleware-Based Syst.*, 2002.

[20] J. Salas, F. Perez-Sorrosal, M. Pati no Martínez, and R. Jiménez-Peris. Ws-replication: a framework for highly available web services. In *WWW*, 2006.

[21] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. Globedb: autonomic data replication for web applications. In *ACM WWW*, 2005.

[22] Spread. homepage: http://www.spread.org/.

[23] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, 2005.

[24] C. Yuan, Y. Chen, et al. Evaluation of edge caching/offloading for dynamic content delivery. In *WWW*, 2003.

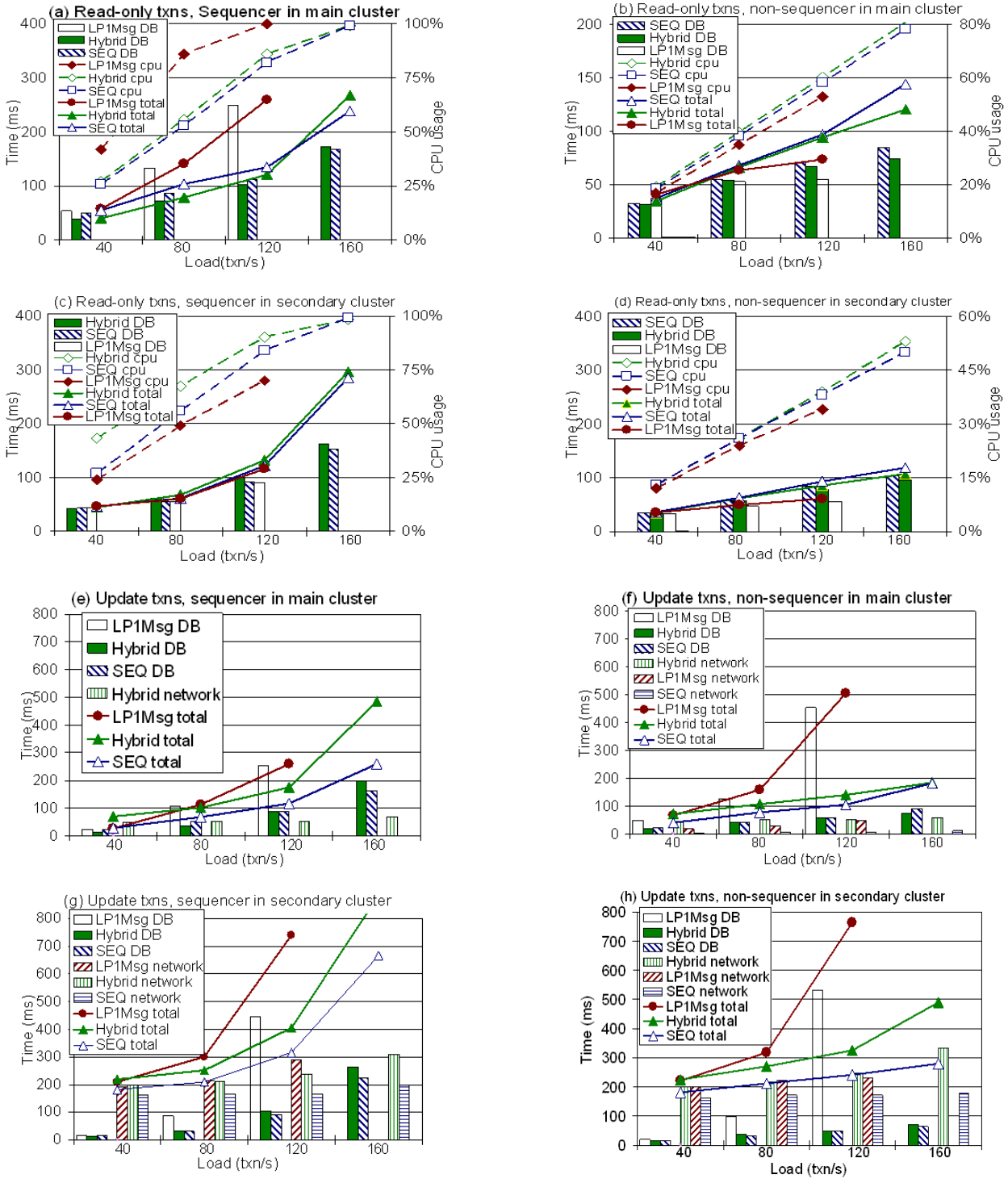

Figure 10. Scenario 2: Clustered edge servers (10 servers) at four cites