Throughput-Driven Database Replication Using a Ring-Based Order Protocol

Ye Liu[1], Paul Ezhilchelvan[1], Yingming Wang[1], and Jim Webber[2]

[1] School of Computing, Newcastle University, NE4 5TG, UK
Email: [Y.Liu197, Paul.Ezhilchelvan, Y.Wang303]@Newcastle.ac.uk
[2] Neo4j UK, London, SE1 0LH, UK
Email: Jim.Webber@Neo4j.com

Abstract. We present a database replication architecture that guarantees ACID transaction properties as well as high throughput expected of modern database systems. Higher throughput results due to server replicas processing distinct, non-overlapping subsets of incoming transactions *in parallel*. Our novel approach addresses all challenges that emerge in ensuring ACID properties across *all* incoming transactions processed in parallel even when access pattern of transactions is not known *a priori*. At the core of our approach is a high-throughput, ring-based total order protocol which the database replicas use to reach consensus for resolving conflicts among transactions, ensuring serializability and accomplishing atomic commit. After presenting the architecture, protocol performance is evaluated through implementations when replication degree is two and three, tolerating at most one replica crash. While 2-fold replication requires perfect crash detection, three-fold can do with weak detectors.

1 Introduction

Replicating a database for high availability has long been studied, implemented, and analysed for various performance characteristics (see [12]). Replication that also ensures ACID properties needs to address greater, additional challenges. *Atomicity (A), Consistency (C), Isolation (I), Durability (D)* are the ACID properties that ensure total integrity at the point of transaction termination, despite host crashes and transactions seeking to access common data items in an order incompatible with C and I properties. For example, the problem of 'incompatible access' or *conflict* becomes more challenging to solve when database replicas process distinct transactions in parallel. Consequently, several performance studies, e.g. [17], shows that ACID replications offer a much smaller throughput even at medium loads, compared to non-ACID ones; however, the latter permit database replica states to diverge and hence require state reconciliation which can be next to impossible in some database contexts [6].

This paper addresses the challenge of improving throughput for ACID replication systems. Our approach involves fully replicating a database on multiple servers that initially execute distinct subsets of input transactions in parallel, but finally generate identical transaction outcomes and commit identical state updates. At the heart of our proposal is a high-throughput ring-based total order

protocol supporting three essentials: event ordering required for server replication, identical inter-replica concurrency control to ensure C and I properties of ACID, and *Two-Phase Commit* (2PC) [9] to guarantee A and D.

While our approach is novel when transaction access pattern is initially unknown, it derives its theoretical underpinning from three canonical findings: (i) total ordering of transactions (or *Atomic Broadcast*) and solving *Consensus* (to resolve conflicting transactions) are reducible to each other under crash failures [2] (ii) the two-phase commit (2PC) is only a simplified instance of consensus [10], and (iii) maximum throughput is achieved when transaction ordering is done over a logical, unidirectional ring network [11].

Consider, as a motivating example, a database replicated on n fail-independent servers: $\{R_i, R_j | 1 \leq i, j \leq n, n \geq 2\}$. Say, concurrent transactions ${T_i}^k$ and ${T_i}^{k'}$ execute in R_i and access overlapping sets of data items. This will be termed as a *local* conflict. To ensure C and I, R_i needs to resolve who waits or gives up for whom, and this can be done autonomously within R_i. Let us say transaction ${T_j}^l$ executes in R_j in parallel and wishes to access replicas of data items common with ${T_i}^k$ executing simultaneously in R_i. This is termed as a *global* conflict which can be detected and be resolved only after parallel executions in R_i and R_j are over. A total order (or simply an order) protocol takes inputs from distributed servers and lets all servers decide identically on an order over the combined set of inputs. It thus provides precedence on transactions for replicas to identically decide which of the globally conflicting transactions waits or gives up.

The paper is organised as follows. After presenting related work below, Section 2 presents our approach together with necessary background. It also presents another popular approach to highlight the novelty of ours. Section 3 describes the role of our order protocol in building a single server abstraction wherein all replicas process all transactions identically (commit or abort) even though they start off processing distinct streams of incoming transactions. Section 4 implements the ring-based protocol and measures its responsiveness. This protocol has been presented earlier in [13] together approximations for response time estimation. The measured results are shown to be close to the estimates, allowing us to dynamically adapt protocol parameters for real world situations. Finally, concluding remarks are provided in Section 5.

1.1 Related Work

A recent book [12] compiles the vast material addressing database replication. The paper [16] classifies replication techniques using three parameters, namely: active vs passive replication, replica interactions per transaction vs per operation, and non-voting vs voting based decision on transaction commit or abort. We use parallelised passive replication with each replica acting as the primary for distinct input streams, per transaction interactions, and non-voting for commit/abort decisions.

The performance study in [17] (referred to earlier) compares five ordering-based replication techniques; lazy replication, where consensus is not applied to resolve conflicts, offers high throughput at a high risk of state divergence for Graph databases as argued in [5]. The order protocol used in [17] for ACID

replication is leader-based where the leader is a performance bottleneck. Our ring-based protocol is leader-free - an important feature that allows it to achieve the highest possible throughput [11].

2 Our approach

2.1 Conflict and Concurrency Control

Concurrent transactions accessing common data items in a database system can create *conflicts*. There are three types of access conflict: after an ongoing transaction, say T_i, has written a data item, say, X, if another one, say T_j, wishes to write or read X, then a *write-write* or *write-read* conflict is said to occur, respectively; similarly, a *read-write* conflict arises when T_j seeks to write X after T_i has read X. Suppose also that T_j has already accessed another data item Y and T_i (after having accessed X) seeks a conflicting access on Y; an incompatibility with C and I guarantees would arise if both T_i and T_j were allowed to go ahead ignoring these access conflicts completely: T_i would have accessed X before T_j with T_j accessing Y before T_i. This would violate C and I which require transactions access *all* common data items in the same order.

The literature proposes many *isolation levels* [14], from the most restricted to unrestricted, to guarantee C and I. The unrestricted *serializability* must eliminate all incompatibility that can arise due to any of three types of conflicts discussed earlier. It will be our target isolation level here.

Irrespective of the isolation level sought, there are broadly two ways to resolve a conflict: *Wait* and *Abort*. In the former, later transaction(s) wait until the earlier transaction completes its execution. Assuming that T_i and T_j are executing in the same replica, T_j will wait to access X until T_i completes while it would be the other way around on Y. So, the wait strategy must be accompanied by deadlock detection and avoidance strategies, e.g., the work from [4] maintain an access graph for ensuring the smallest possible abort ratio.

In *Abort* approach, one transaction (T_i or T_j) aborts itself after a limited or no waiting. We use here *No-Wait, Instantaneous Abort* wherein a transaction encountering a conflict will instantly abort itself. Thus, if both T_i and T_j are running on the same replica, their access conflicts would be *local* and can lead to both aborting as they individually encounter a conflict. Contrary to the intuition that *Instantaneous Abort* may cause too many transactions to abort, our earlier evaluations (see [7]), both model- and implementation-based, demonstrate that the abort ratio is acceptably small and not considerably larger than that the smallest returned by [4]. *Instantaneous Abort*, on the other hand, eliminates deadlock possibilities, extracts near-zero implementation overhead in replicated systems and also is the most throughput-friendly.

If T_i and T_j are running on different replicas, say in R_i and R_j respectively, their conflicts would be *global* and not detectable during their parallel execution. Post-execution, if the order protocol orders, say, T_i before T_j then all n replicas will deem T_j to be aborted and accept the outcomes of T_i execution in R_i.

2.2 Architectures of Replicated Database System

We present our architecture after presenting one of the most popular replication architectures found in the literature and adopted by many practitioners including Google. Thereby, we seek to highlight the novelty of our architecture in incorporating parallel processing to promote throughput. The common architecture will be referred to as *Replication Architecture 1*, or simply as **RA1**, and ours as **RA2**. They are depicted in Figures 1 and 2 respectively.

In both RA1 and RA2, clients submit transactions directly to only *one* of the n replicas, *R1*, *R2*, ..., *Rn*; in RA1, there is no parallel processing. More specifically, all replicas first exchange with each other the transactions they directly received, and then order them all identically, prior to processing each of them. That is, all replicas *actively* execute all transactions in the same order. Thus, there are no global conflicts and conflicts encountered are resolved using the same concurrency control (CC) mechanism based on the transactions order.

Barring race conditions, if one replica aborts a given transaction then all would do so. Disagreements due to race conditions and effects of crashes are dealt with during 2PC execution, with each replica acting as the 2PC leader for transactions that it directly received. Spanner [3], Megastore [1] and CockroachDB [15] are recent systems that have adopted RA1. Spanner use TrueTime and others the (leader-based) Paxos or Raft protocols for ordering.

In **RA2** (see Figure 2), replicas execute the transactions they directly received, in parallel, and using some local concurrency control (CC) mechanism. They then exchange, for each locally survived (i.e., not aborted) transaction, the local transaction identifier and a list of data items accessed and the current value of each write-accessed data, using an order protocol. Global conflicts are detected using data access information and resolved using the order decided on locally survived transactions. Those that survive global conflict resolution proceed to 2PC to be committed. In Figure 2, $T_1^n, T_3^n, \ldots$ shows a sub-stream of locally survived transactions emerging from R_n, and $T_1^1, T_1^2, T_3^n, \ldots$ show globally survived transactions emerging identically from all replicas.

RA2 is our architecture and uses our ring-based ordering protocol. As noted earlier, any order protocol enables each replica (i) to disseminate its set of locally survived transactions to other replicas, and (ii) to decide an identical order on the combined input super-set. We exploit the former aspect to simplify 2PC implementation by piggybacking relevant information.

Note, however that the traditional 2PC is a leader-based protocol and our order protocol is leaderless. So, 2PC needs to be appropriately adapted. The next two subsections provide the background for this adaptation which is detailed in Section 3.1.

2.3 Ring-based Order Protocol

Our ring-based order Protocol that has been proposed and modelled for performance in [13], is implemented here for $n = 2$ and 3. Figure 3 shows n database replicas being arranged in a logical ring and a *Folder* continually circulating in clockwise direction: moving from Replica R_1 to R_2, R_2 to R_3, ..., R_n to R_1, and

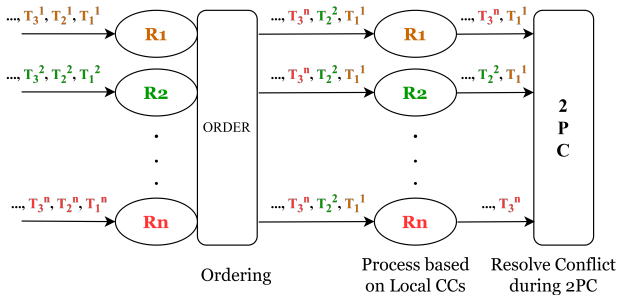

Fig. 1: RA1 - Active Replication: Ordered Processing Everywhere

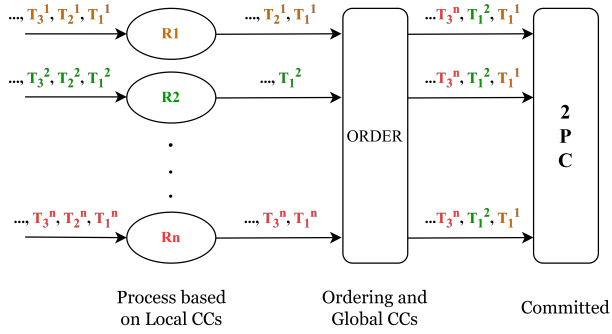

Fig. 2: RA2: Order-based Global Conflict Management before Commit

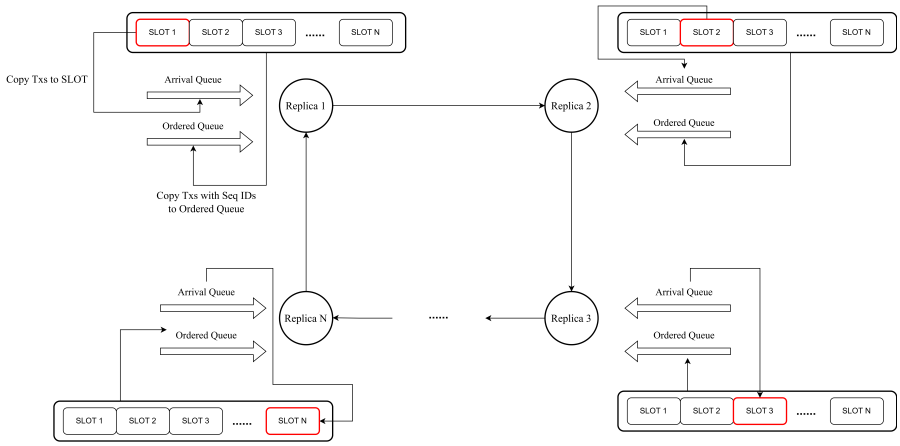

Fig. 3: Ring-Based Order Protocol: Unidirectional Folder Circulation

so on. The folder contains one slot of fixed size for each replica. Each replica R_i enqueues each locally-survived and completed transaction in the *Arrival Queue* (see Fig. 3) together with a list of data items accessed and the final values of data items written. When it receives the folder, it enqueues all transactions found in the folder in the *Ordered Queue* as per their sequence number, empties its own slot in the folder, dequeues a transaction from *Arrival Queue* and loads it into its slot with a sequence number that is one more than the largest found in the folder. This loading continues until *Arrival Queue* is empty or there is no more space left in its slot. When it stops, folder is sent to R_{i+1} or R_1 if $i = n$.

2.4 Two-Phase-Commit Protocol (2PC)

2PC [8] protocol involves two phases orchestrated by a lead replica, called the *coordinator* and its execution for any given transaction results in either all replicas committing or aborting that transaction. In the first, *prepare* phase the coordinator sends a *prepare* request to all replicas, asking whether they can successfully commit the transaction. Each replica evaluates commit feasibility by checking for constraint violations and locks write-accessed data items. If a replica can commit, it responds affirmatively; otherwise, negatively. If all respond affirmatively, the coordinator advances to *commit* phase by instructing the replicas to commit the transaction; even if one replica sends a negative or no response (due to crash), the coordinator sends 'abort' to all replicas which abort that transaction. This original 2PC version is adapted in Section 3.1 to be executed at the back of the circulating folder in a coordinator-free manner.

3 Ring-Based Ordering & 2PC for 1-Server Abstraction

1-Server abstraction in crash-tolerance literature refers to multiple, crash-prone server replicas 'coming together' to provide a client-level abstraction of a single, crash-free server provided that replica crashes do not exceed a specified threshold. We explain here how this coming-together is realized through our ring-based order protocol using, for simplicity, $n = 2$ replicas of which at most one can crash. Recovering from a crash in a ring structure is discussed in detail in [11], so we focus here on crash-free operation by referring to Figure 4 where data structures and folder slots of R_1 and R_2 are shown in red and blue respectively.

Replica R_i, $i \in \{1, 2\}$, executes the transactions that it directly receives. If a transaction T_j needs to access data item X that has already been accessed by an ongoing, concurrent T_i, then T_j will instantly abort. Transactions that complete their executions without being aborted will have their identifiers together with their *data access list*s queued in the local *Arrival Queue*, AQ_i for short, and also entered in *Local Survived list*, $LS - List_i$ (see Fig. 5). A transaction identifier is the one generated within a replica, concatenated with the replica's id; it is therefore unique within the replicated system, e.g., T_i in R_i can become as ${T_i}^i$ (see also Fig. 2). The *data access list* for ${T_i}^i$, denoted as $DAL({T_i}^i)$ is the list of all data items that ${T_i}^i$ accessed and the final values of write-accessed data items. For example, if ${T_i}^i$ read-accessed X and Z and left the write-accessed Y with value $Y = 10$ on completion, then $DAL({T_i}^i)$ will be $\{X, Y = 10, Z\}$.

Whenever R_i receives the circulating, two-slotted folder, it notes down the largest sequence number in the folder and copies all entries in each slot of the folder into its *Ordered Queue*, OQ_i for short, as per their sequence number. It then empties its own slot in the folder and then loads as many AQ_i entries into its slot, with each loaded entry assigned a sequence number continuing sequentially from the largest noted. If, say, the latter is 110 and R_i loads nine entries, their sequence numbers would be $111, 112, \ldots, 119$. Once its slot is loaded to its capacity, the folder is sent to the next replica in the ring, $R_j, j \neq i$; when R_j receives the folder, it would note the largest sequence number as 119.

Two remarks are in order. First, when R_i is in possession of the folder, it does not modify the contents of any slot other than its own. Secondly, contents of all slots of the arriving folder are copied for ordering, including its own slot which was loaded when R_i had the folder in the previous cycle; those being loaded now would be copied and entered into OQ_i when the folder returns next. It is easy to see that replicas enter all locally-survived transactions into their respective OQ in the same (sequence number based) order.

Whenever OQ_i is non-empty, R_i dequeues the first item and compares the DAL in the dequeued item with the DAL of *every* entry stored in the *Global-Survived List*, $GS - List_i$, that it maintains (see Fig. 5). If a conflict is detected with any $GS - List_i$ entry, the dequeued item is entered in *Global-Aborted List*, $GA - List_i$ and the corresponding transaction must be aborted during 2PC; otherwise, the dequeued item is entered in $GS - List_i$ for commit during 2PC.

Let us consider an example: let $\{{T_i}^i, DAL({T_i}^i)\}$ be the item freshly dequeued from OQ_i and let $\{{T_h}^j, DAL({T_h}^j)\}$ be some entry in $GS - List_i$ which would refer to transaction T_h that locally survived in R_j and was sequenced before ${T_i}^i$ and hence is already in $GS - List_i$. If $DAL({T_h}^j)$ indicates that ${T_h}^j$ wrote data item X when it was executed at R_j and $DAL({T_i}^i)$ indicates that ${T_i}^i$ read data item X when it was executed in parallel at R_i, then it would be treated as a global *write-read* conflict and the later ordered ${T_i}^i$ is marked for abort.

All replicas will reach the same outcome while checking a given item dequeued from their respective OQ *provided* that their respective *GS-List*s also have identical contents at the time of their checking. The latter cannot be guaranteed as globally-survived transactions that have gone through the next stage 2PC execution must be removed from *GS-List*. Due to inherent asynchrony in distributed computing, replicas may decide differently: for example, R_i entering $\{{T_i}^i, DAL({T_i}^i)\}$ in $GS - List_i$ and R_j entering the same $\{{T_i}^i, DAL({T_i}^i)\}$ in $GA - List_j$ instead. This can occur, for example, if ${T_h}^j$ had been removed from $GS - List_i$ following its commit in R_i but is yet to be removed from $GS - List_j$ at the time when R_i and R_j dequeued $\{{T_i}^i, DAL({T_i}^i)\}$ from their respective *GS-List*. This inconsistency will be sorted out during 2PC when R_j would respond negatively for committing ${T_i}^i$ and force R_i also to abort ${T_i}^i$. Thus, all replicas either commit or abort a given input transaction irrespective which replica directly executed that transaction; i.e., 1-server abstraction is ensured.

3.1 2PC on the Back of Ring-based Ordering

To implement 2PC using the circulating folder, we require that the folder, in addition to n slots, contains a list of *Blocks*, one *block* for each *transaction* that has entered the *GS-List* and hence is ready to be committed through 2PC execution. As noted in Subsection 2.4, all replicas must respond affirmatively after *prepare* phase for a commit outcome and a replica's 'no' acts as a 'veto'.

The structure of a transaction's *block* reflects this information collection: transaction (global) identifier (e.g., T_h^j) accompanied by an integer vector of n indices - one for each replica (see Fig. 5 for $n = 3$). The vector in the block for transaction T is denoted as V_T. $V_T[i], 1 \leq i \leq n$, is 0, 1 and 2 respectively implies that R_i possibly started 2PC for T and is *preparing*, R_i completed *prepare* phase and is ready for *commit* phase, and R_i committed T; $V_T[i] = -1$ indicates that R_i responded negatively for T in *prepare* phase and therefore $V_T[j]$ can never become 2 for any replica R_j.

Each R_i also maintains four more lists (shown in Fig. 5) and their abbreviated names follow this intuitive convention: **S** for Survived, **L** for Local, G for Global, and **P, C** and **A** for Prepared, Committed and Aborted respectively.

For space reasons, we explain the workings of our *Ring-Based 2PC* for transaction T by referring to Fig. 6 where it is assumed that all replicas enter T in their respective *GS-List*. The other case of only some replicas doing so and others entering T in *GA-List* is simpler and discussed next. We also assume that a replica optimistically starts preparing for committing T (i.e., the *prepare* phase) as soon as it enters T in its *GS-List* and similar optimism is common in 2PC implementations; *Replica i*, $1 \leq i \leq 3$, in Fig. 6 is simply referred to as R_i.

The description based on Fig. 6 involves 11 Steps; transition to the next step corresponds to a given replica receiving the folder circulating in the ring.

Step 1: We assume that *Replica 1* (R_1) is the first replica to notice that the folder does not yet have a block for T that is in its $GS - List_1$. So, it adds a block for T in the *Blocks* part of the received folder with $V_T =$ [0,0,0]. Suppose that preparation for T is locally completed; $V_T =$ is set to [1,0,0] and the entry for T in $GS - List_1$ is moved to *Locally Prepared List*, $LP - List_1$. (If not completed, $V_T[1]$ is left unchanged at 0 and these operations are to be carried out at the earliest instance when R_1 receives the folder after it has completed *prepare* for T.) When R_1 is done with the folder, it transmits the folder to R_2.

Step 2: Suppose that R_2 has also completed *prepare* for T when it receives the folder. It sets $V_T =$ to [1,1,0] and moves the entry for T from $GS - List_2$ to $LP - List_2$. When ready, R_2 will transmit the folder to R_3.

Step 3: Suppose also that R_3 also completed *prepare* phase for T when it receives the folder (from R_2). It will behave like *Replica 2* stated in Step 2, except that $V_T =$ [1,1,1] and R_3 now deduces that all 3 replicas have completed *prepare* for T, starts the next 2PC phase *commit* for T and moves the entry for T from $LP - List_3$ to *Global-Prepared List*, $GP - List_3$.

Step 4: When R_1 receives the folder with $V_T =$ [1,1,1], it deduces that all replicas have done *commit* for T; so, it moves the entry for T from $GP - List_1$ into $GP - List_1$, *Global-Prepared List* and starts *commit* for T.

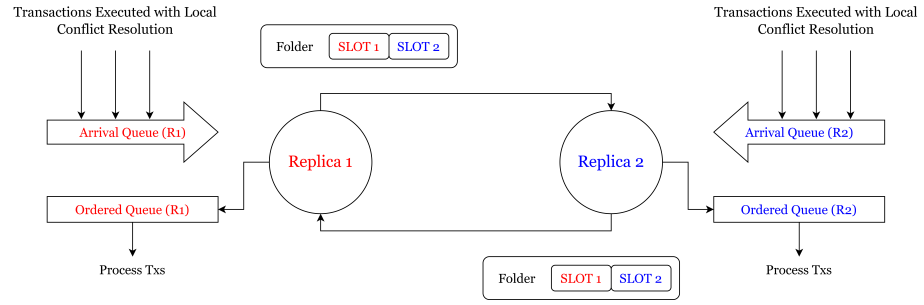

Fig. 4: Ordering Locally-Survived Transactions Using Circulating Folder.

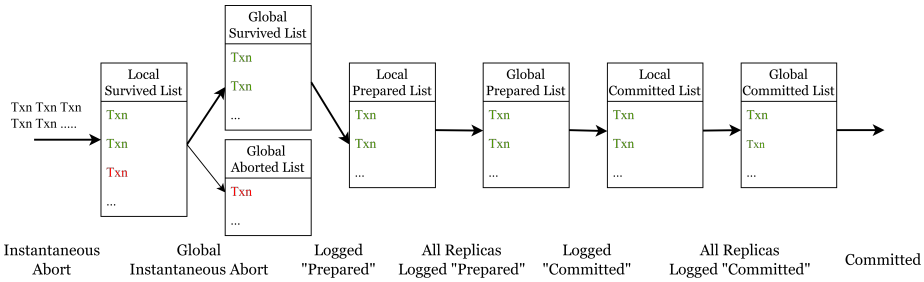

Fig. 5: Lists for Transaction Lifecycle Management

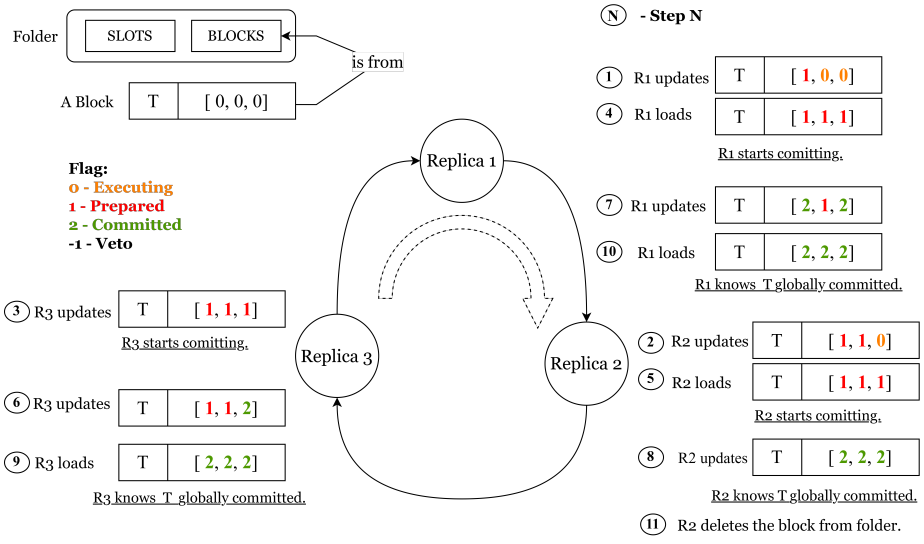

Fig. 6: Ring-Based 2PC Steps in Committing Transaction T

Step 5: R_2, on receiving the folder, starts *commit* for T, like R_1 in Step 4.
Step 6: Recall that R_3 started *commit* for T in Step 3. It is possible, and we here assume so, that when R_3 receives the folder now, it has completed *commit* for T; if so, it will move the entry for T from $GP - List_3$ to $LC - List_3$ and sets $V_T = [1, 1, 2]$. (If *commit* is not completed, these actions will be done at the earliest instance when R_3 receives the folder after *commit* for T is locally completed.)
Step 7: Suppose that R_1 also completed *commit* for T when it receives the folder now. It sets $V_T = [2, 1, 2]$ and moves entry for T from $GP - List_1$ to $LC - List_1$, the *Local Committed List* (see Fig. 5).
Step 8: Retaining the assumption that a replica completes *commit* for T when the folder makes one round in the ring, R_2, when it receives the folder, sets $V_T = [2, 2, 2]$ and moves entry for T from $GP - List_2$ to $LC - List_2$. Additionally, it deduces from $V_T = [2, 2, 2]$ that T is committed everywhere and moves entry for T from $LC - List_2$ to $GC - List_2$.
Step 9: R_3 locally carries out the additional operations done by R_2 in Step 8.
Step 10: R_1 locally carries out the operations done by R_3 in Step 9. Note: all replicas now have T in their $GC - List$.
Step 11: *Garbage Collection Rules* are as follows. Whenever R_i receives the folder with V_T already set to $[2, 2, 2]$ for any T, it deletes the entry for T in $GC - List_i$ if it *already* exists in $GC - List_i$; otherwise, it deletes the block for T in the *Blocks* part of the incoming folder. So, R_2 discards its $GC - List_2$ entry for T here when folder arrives with $V_T = [2, 2, 2]$.

Continuing on, R_3 and R_1 will discard their $GC - List$ entry for T in Step 12 and 13 respectively, and R_2 will discard the *Blocks* entry for T in Step 14.

Executions with $T \in GA - List_i$ for some or all $R_i, 1 \leq i \leq 3$. Suppose that R_1 has entered T in its $GA - List_1$ and receives the folder in Step 1 above with no *Blocks* entry for T. It will create an entry with $V_T = [-1, 0, 0]$ Any replica R_j that receives the folder sees this veto for T will be in one of *two* situations: (i) it also has T in $GA - List_j$ in which case it sets $V_T[j]$ to -1, or (ii) it has T in $GS - List_j$ in which case it aborts any *prepare* done or being done for T, moves the $GS - List_j$ entry for T into $GA - List_j$ and sets $V_T[j] = -1$.

Another scenario of interest is that R_1 has entered T in its $GS - List_1$ and behaves as in Step 1 above, and only replica that has entered T in its $GA - List$ is R_2 and/or R_3. Let us assume that it is only R_2 with T in $GA - List_2$. In Step 2, R_2 will execute the actions of (i) above and R_3 and R_1 will execute the actions of (ii) above in Step 2 and 3 respectively. *Garbage Collection Rules* are as before except that the incoming folder should have V_T already set to $[-1, -1, -1]$ (instead of $[2, 2, 2]$) and $GA - List$ is used instead of $GC - List$. Thus, R_2, R_3 and R_1 will discard their $GA - List$ entry for T in Step 5, 6 and 7 respectively; in Step 8, R_2 will discard the *Blocks* entry for T in the received folder.

4 Implementation and Performance Evaluation

We implemented the ring-based order protocol with $n, n = 2, 3$, servers forming the ring and measured its performance in terms of *Latency* that is defined as

the time elapsed between the instance a locally-survived transaction enters the *Arrival Queue (AQ)* of a replica and the moment it has been entered in the *Ordered Queue* of all replicas (see Fig 3), the given slot size k is 1KB in the *Folder* for every replica. *Latency* thus includes the wait-time in AQ and the time spent in the circulating folder.

We compared the uniform *Measured Latency* with the *Estimated Latency* from model-based analytical approximations presented in [13]. While the former is measured by repeating experiments on the implemented system, the latter requires measuring two system parameters: the average time (α) taken by replicas to process a received folder and the average folder transmission time (β) between replicas on the ring; measurements found $\alpha = 10^{-3}$ secs and $\beta = 10^{-5}$ secs.

Analysis in [13] states that the system is stable when the average arrival rate into AQ, denoted as λ, is: $\lambda < \frac{k}{n^2 k\alpha+\beta}$, where k is the number of slots in the folder which is n. Since β is negligibly small compared to $n^2 k\alpha$, we can conclude that the largest stable λ is inversely proportional to n^2 which will guide our choice of λ values for experiments. Latency is estimated as follows.

For any stable λ, a unique solution for $s \in (0, 1)$ exists:

$$s = \frac{\lambda(n\alpha + \beta)}{1 - \lambda n(n-1)\alpha} \ . \tag{1}$$

The average queue size L at each replica is estimated as $\frac{s}{1-s}$ and the estimated Latency (by Little's Law) is $\frac{L}{\lambda}$.

4.1 Two Replica System

With $n = 2$, the maximum stable λ is to be no larger than 249. We began by measuring the Latency under varying arrival rates, specifically at $\lambda = 50$, 100, 150, 200, and 249. Starting from $\lambda = 200$, a notable increase in both the measured and estimated L values is observed, indicating a sharp transition in system behaviour. At $\lambda = 249$, the values become extremely large, reflecting system is approaching instability or saturation.

We conducted additional experiments focusing on the critical range $239 \leq \lambda \leq 249$, collecting a more granular set of data to understand the system dynamics in this region better. Detailed experimental results for $239 \leq \lambda \leq 249$, including the estimated latency, are summarized in Table 1.

Our experimental results reveal that the system achieves its optimal performance when the transaction arrival rate λ lies under 247. When λ approaches 248 or 249, the system performance degrades noticeably but remains operational. However, when λ exceeds 249, the system surpasses its capacity limits, leading to system instability.

Finally, as illustrated in Figure 7, the measured latency values closely align with estimated ones, indicating that the system operates in a stable and predictable regime. This consistency confirms that $\lambda \leq 247$ represents a robust region for maintaining high throughput with bounded latency.

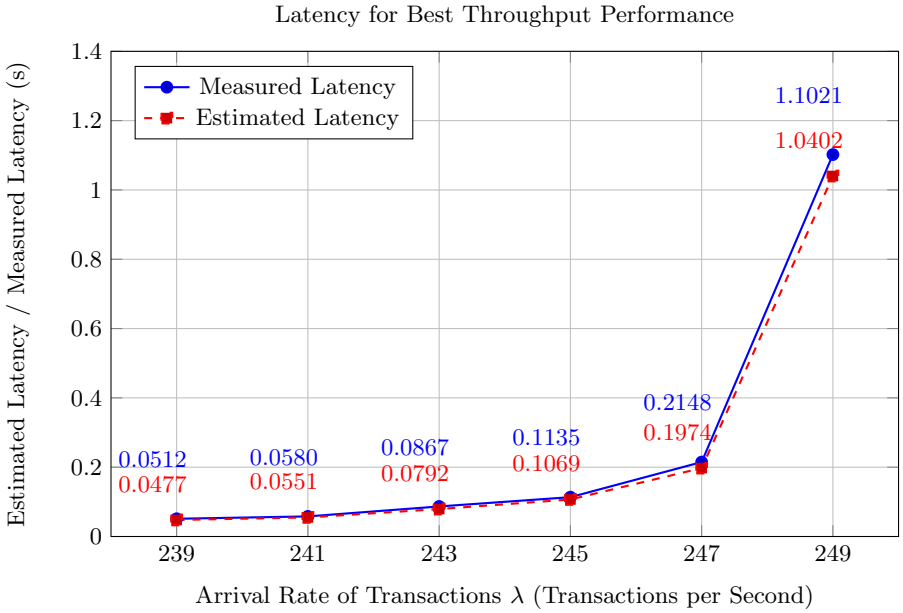

Fig. 7: Two Replicas with Increasing λ. $\lambda \in [239, 249]$

λ	**Measured L**	**Estimated L**	**Estimated Latency (s)**	**Measured Latency (s)**
239	11.3890	11.5451	0.0477	0.0512
241	13.2804	14.4213	0.0551	0.0580
243	19.2459	19.1017	0.0792	0.0867
245	26.2024	28.0598	0.1069	0.1135
247	48.7488	52.0955	0.1974	0.2148
249	259.0055	331.4503	1.0402	1.1021

Table 1: Two Replicas: Results for λ in the Range 239 to 249

λ	**Measured L**	**Estimated L**	**Estimated Latency (s)**	**Measured Latency (s)**
20	0.0712	0.0734	3.56×10^{-3}	3.73×10^{-3}
40	0.1774	0.1882	4.435×10^{-3}	4.521×10^{-3}
60	0.3686	0.3931	6.143×10^{-3}	6.495×10^{-3}
80	0.8119	0.8625	1.015×10^{-2}	1.105×10^{-2}
100	2.8382	3.04	2.838×10^{-2}	2.897×10^{-2}
110	32.6624	37.2022	0.2969	0.3175

Table 2: Three Replicas: Results for λ in the Range 20 to 110

4.2 Three Replicas: $n = 3$

Following the same analytical framework, when the number of replicas increases to $n = 3$, the system's stability threshold for the maximum permissible value of λ decreases notably. The results indicate that λ must be constrained to be no larger than 110 to maintain stable operations under this configuration. Similar to the case with two replicas, the parameter k continues to have a negligible impact on system stability, reinforcing the robustness of the protocol design against variations in transaction size or queue depth at this scale.

Table 2 shows the experimental results for a system with three replicas ($n = 3$), evaluated under varying transaction arrival rates λ ranging from 20 to 110. The Table presents the measured and estimated values of L, representing the average number of transactions in the system, as well as the corresponding measured latency and measured delay values.

The data shows that the system load increases correspondingly as λ increases. The measured and estimated L values are closely aligned across all tested λ values, indicating that the analytical model accurately estimates the average system load under varying traffic intensities.

In terms of latency performance, the estimated latency and measured latency also exhibit a strong correlation. Both metrics increase as λ grows, which is consistent with queuing theory expectations. For lower λ values (e.g., $\lambda = 20$), the system maintains very low latency, with delays in the millisecond range. As the transaction arrival rate increases towards $\lambda = 110$, both latency and delay reach sub-second values, highlighting the impact of increased load on system responsiveness.

As λ increases, the system load and message delay also increase predictably, following expected queuing behaviour. The close alignment between theoretical predictions and empirical results validates the accuracy and robustness of the proposed analytical model. Even under higher traffic conditions, the system maintained consistent and stable behavior, suggesting that the message ordering protocol scales effectively in distributed environments with multiple replicas.

5 Conclusion

The Ring Based Ordering and 2PC Protocols provide an effective solution for database replication while ensuring the most desirable ACID properties which eliminate state divergence and the need for complex reconciliation mechanisms. Moreover, by allowing parallel processing at replicas, a higher throughput is maintained. Thus, our ring based replication architecture represents a significant step towards high-performance, fault-tolerant database replication, providing a novel, robust foundation for modern distributed systems.

Our experiments on the implemented system show that actual performance is remarkably close to our analytical estimations. An advantage of this closeness is that we can use our model-based estimations to adapt the system for prevailing arrival rates, e.g., by increasing the number of slots per replica. We are currently designing order protocols with multiple circulating folders. Our initial assessments show that even higher throughputs are possible but at the cost of

increased latency. Thus, the single-folder protocol presented here represents one end of a range of ring-based options that can be exercised.

References

1. Jason Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.
2. Tushar Deepak Chandra et al. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.
3. James C. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), August 2013.
4. Dominik Durner and Thomas Neumann. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 734–745. IEEE, 2019.
5. Paul Ezhilchelvan, Isi Mitrani, and Jim Webber. On the degradation of distributed graph databases with eventual consistency. In *Computer Performance Engineering*, pages 1–13. Springer, 2018.
6. Paul Ezhilchelvan, Isi Mitrani, and Jim Webber. Modeling the gradual degradation of eventually-consistent distributed graph databases. *Queueing Models and Service Management*, 3(2):235–253, 2020.
7. Paul Ezhilchelvan, Isi Mitrani, Jim Webber, and Yingming Wang. Evaluating the performance impact of no-wait approach to resolving write conflicts in databases. In *European Workshop on Performance Engineering*, pages 171–185. Springer, 2023.
8. James N Gray. Notes on data base operating systems. *Operating systems: An advanced course*, pages 393–481, 2005.
9. Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, page 393–481, Berlin, Heidelberg, 1978. Springer-Verlag.
10. Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.
11. Rachid Guerraoui et al. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems (TOCS)*, 28(2):1–32, 2010.
12. B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis lectures on data management. Morgan & Claypool Publishers, 2010.
13. Ye Liu, Paul Ezhilchelvan, and Isi Mitrani. Design and analysis of distributed message ordering over a unidirectional logical ring. In *European Workshop on Performance Engineering*, pages 1–13. Springer, 2024.
14. Jim Melton. Ansi/iso sql-92 specification. Online, 1994.
15. Rebecca Taft et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.
16. M. Wiesmann et al. Database replication techniques: a three parameter classification. pages 206–215, 2000.
17. M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.