Database Replication: a Tale of Research across Communities

Bettina Kemme
Department of Computer Science
McGill University
Montreal, Canada
kemme@cs.mcgill.ca

Gustavo Alonso
Systems Group
Department of Computer Science, ETH Zurich
Zurich, Switzerland
alonso@inf.ethz.ch

ABSTRACT

Replication is a key mechanism to achieve scalability and fault-tolerance in databases. Its importance has recently been further increased because of the role it plays in achieving elasticity at the database layer. In database replication, the biggest challenge lies in the trade-off between performance and consistency. A decade ago, performance could only be achieved through lazy replication at the expense of transactional guarantees. The strong consistency of eager approaches came with a high cost in terms of reduced performance and limited scalability. Postgres-R combined results from distributed systems and databases to develop a replication solution that provided both scalability and strong consistency. The use of group communication primitives with strong ordering and delivery guarantees together with optimized transaction handling (tailored locking, transferring logs instead of re-executing updates, keeping the message overhead per transaction constant) were a drastic departure from the state-of-the-art at the time. Ten years later, these techniques are widely used in a variety of contexts but particularly in cloud computing scenarios. In this paper we review the original motivation for Postgres-R and discuss how the ideas behind the design have evolved over the years.

1. INTRODUCTION

Data replication is a fascinating topic for both theory and practice. On the theoretical side, many strong results constraint what can be done in terms of consistency: e.g., the impossibility of reaching consensus in asynchronous systems [16], the blocking nature of 2PC [18], the CAP theorem [8], and the need for choosing a suitable correctness criterion among the many possible [7, 42]. On the practical side, data replication plays a key role in a wide range of contexts: caching, back-up, high availability, wide area content distribution, increasing scalability, parallel processing, etc. Finding a replication solution that is suitable in as many such contexts as possible remains an open challenge. In recent years, database replication has acquired an additional dimension due to the role it plays in achieving elasticity when combined with virtualization in cloud computing environments (the most recent example being the architecture of SQl Azure [9] or proposals to support multi-tenancy in databases [37]).

In this paper we review the original results of the VLDB 2000 paper that presented Postgres-R [23]. Postgres-R was the first database replication system that was both scalable and provided strong consistency guarantees. It fulfilled these two design goals by taking a completely different approach to implementing replication than the commercial systems and research proposals available at the time. The main contributions of Postgres-R covered a wide range of aspects but complemented each other very well: dealing with consistency outside the database; considering alternative correctness criteria that reduced overhead but still guaranteed consistency; and showing how to make these new ideas work within a database engine by providing a full implementation. These innovations were possible only by stepping out of the constrained thinking of databases and distributed systems at the time. Postgres-R emerged from the combination of results from both areas, with the ideas and concepts from Postgres-R that have survived the test of time being mainly those that resulted from such synergy.

The most important innovation in Postgres-R was the use of ideas from distributed computing (group communication) to solve the problem of coordinating updates at several copies while maintaining overall consistency [22]. Nowadays, when protocols like Paxos [24] are a core component in many distributed systems, the idea may seem obvious. At the time, it was quite a counterintuitive design choice in both the database as well as in the distributed systems communities. On the one hand, group communication was developed almost exclusively for fault tolerant purposes and Postgres-R used it mainly for performance reasons (as a way to reduce the cost of achieving consistency). On the other hand, understanding how group communication interacted with transaction management in a database required to go away from the fully synchronous model of replicated databases (based on distributed locking and 2 Phase Commit) and adopting a less tightly coupled approach based on ordering guarantees. Today, group communication (or some form of agreement protocol) is widely used as a way to implement database replication in both commercial systems and research projects.

A second contribution of Postgres-R was to exploit the use of multiple data versions to simplify concurrency control (i.e., shadow copies or multi-version concurrency control).

Proceedings of the VLDB Endowment, Vol. 3, No. 1

The version of PostreSQL we used to implement Postgres-R only supported strict 2 Phase Locking (2PL) and, thus, Postgres-R provided serializability. Yet, Postgres-R went to great lengths to implement a rudimentary multi-version system within PostgreSQL due to the many advantages it offered. This optimization opened the way to using more suitable correctness criteria for database replication [22], something that became easier to do a few years later due to the increasing availability of *snapshot isolation* in relational database engines. Interestingly enough, the work on exploring the best way to combine group communication with transactional correctness [22] found resonance mainly among researchers in distributed systems, achieving only limited visibility in the database area.

The third contribution of Postgres-R was a full implementation in a real database system, taking advantage of and accepting the trade-offs that come with the use of a specific database kernel. Fine-tuning all the engineering aspects of working within a relational engine turned out to have a significant impact on performance and was determinant in convincing others that the approach could be made to work. However, and highlighting the differences between the two research communities, the implementation seemed to be relevant mainly to researchers in databases, being clearly of much less interest to the distributed systems community.

Taking advantage of the perspective gained a decade later, in this paper we review the architecture of Postgres-R pointing out what ideas proved to be the right ones and what aspects of the design are now obsolete. In the paper we will put special emphasis on the interplay between databases and distributed systems in the conviction that today, like a decade ago, much is to be gained by combining existing results from the two areas in novel and innovative ways -even if it is a difficult exercise. For reference, we also provide a brief overview of how the initial ideas behind Postgres-R have evolved and are being used in the various systems that have adopted and adapted them.

2. REPLICATION 10 YEARS AGO

2.1 Background

Working on finding a way around the *dangers of replication* [17] was not the most popular research topic in databases ten years ago. Not so in distributed systems where the relation between database replication and agreement problems had been explored in some detail (e.g., [39, 19]). The view held in the database community was that one could get either performance by sacrificing consistency (*lazy replication* approaches) or consistency at the cost of performance and scalability (*eager replication* approaches). Furthermore, the choice seemed to have been made already: performance and scalability took precedence over consistency[1] with most replication solutions using a primary copy approach where changes were propagated after commit and no guarantees were given on the data read from the copies. The fact that it was possible to implement (and commercialize!) a system with such ill-defined consistency guarantees was always met with incredulity by the distributed systems community. In the database world, the protocols used in distributed systems, and specially group communication, were perceived as complex and expensive without actually solving a real database problem.

[1]A situation resembling today's discussions around relational database engines, the NoSQL movement, and elastic data processing in the cloud.

2.2 The Dangers of Replication

Ten years ago, the theoretical basis for database replication revolved around the classic concepts of serializability and locking [7, 42]. Replication was implemented with *one-copy-serialazability* as correctness criterion (a history would produce the same results as an equivalent history over a system with a single copy). This was achieved through a *read-one-write-all* protocol where read operations will obtain local locks and write operations will obtain distributed locks on all copies. Atomicity was ensured by using a *2-Phase-Commit* (2PC) protocol at the end of each transaction. Quorums were often suggested as a way to minimize overhead by not having to modify all copies in each transaction but only a subset of them. The result was a fully consistent system that behaved like a single database.

While elegant and relatively easy to understand, such an approach to replication hided many pitfalls and complex engineering aspects. Gray et al. [17] were the first to point out some of them. In particular, Gray et al. emphasized that such an approach was based on coordinating each operation individually. As a result, when the number of copies increased, the transaction response times, the conflict probability, and the deadlock rates would grow exponentially. The response times increased because of the overhead of distributed locking (involving messages over the network for each write operation) and the need to run 2PC to commit the transaction (another two rounds of network messages per transaction). Since transactions lasted longer, they locked items for a longer time, thereby increasing their conflict profile and the probability that they blocked other transactions. This resulted in an exponential deadlock rate and dismal response times as the number of copies increased. Based on these observations, Gray et al. concluded that the, at the time, textbook approach to database replication could not possibly scale and proposed a number of alternative lazy approaches.

Because of these results and the prevalence of lazy replication solutions in commercial systems, research on replication before Postgres-R was mostly focused on understanding the inconsistencies created by lazy replication strategies: weak consistency models, epidemic strategies, imposing restrictions on the placement of copies, using a primary copy approach, and hybrid solutions providing consistency within transactional boundaries but propagating updates only lazily (see the original Postgres-R paper for references and a more detailed discussion of these approaches).

2.3 Replication in Practice

At the time we developed Postgres-R, there was only one major commercial database that supported consistent replication. The *Oracle Advanced Replication* solution worked by first performing updates locally and then using *after-row* triggers to propagate those changes synchronously to the other copies. There was no pretense that the approach would scale up and the manuals said so explicitly. In fact, it was provided mostly as a way to implement hot-standby backup systems.

When we analyzed in detail the behavior of such a system, the results correlated quite well with the observations

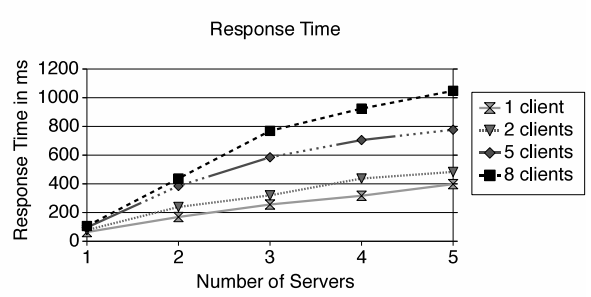

Figure 1: Response time in conventional replication

of Gray et al. [21]. Figure 1 shows the performance exhibited by such conventional approach to replication under a relatively small load of 10 update transactions per second. The response time increased significantly with both the number of copies and clients connected to the system, clearly indicating that replication was not scaling with either higher loads nor with higher multi-programming levels.

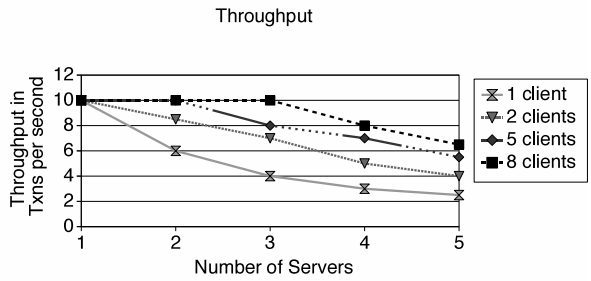

Figure 2: Throughput in conventional replication

Figure 2 shows the corresponding throughput of the system. The drop in throughput caused by replication is also clearly visible, with an extreme case for the single client experiment where a single copy can run 10 transactions per second but 5 copies could only run about 2 transactions per second due to the overhead of synchronization.

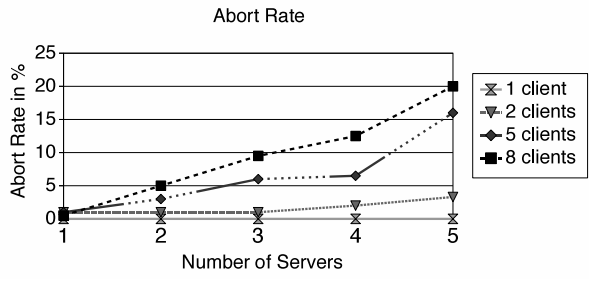

Figure 3: Abort rates in conventional replication

Finally, Figure 3 shows the abort rate. Due to the increase in the conflict profile of the transactions, the abort rate grew very quickly with the multi-programming level, a clear sign that the system did not scale and fully confirming the observations of Gray et al.

These experiments provided very well defined design constraints for Postgres-R: avoid coordinating on a per operation basis; avoid running 2PC for each transaction; and shorten the length of the transactions to minimize their conflict profile.

2.4 Group Communication

Parallel and independently to the work on database replication, a wide range of group communications systems had been developed and studied in great detail in the distributed systems and distributed computing communities [30, 29, 6]. Unlike in databases, where replication served and still serves a wide range of purposes from availability to performance and elasticity, the purpose for replication in distributed computing is almost exclusively fault tolerance with performance rarely playing any role in the design of the corresponding algorithms.

Group communication systems manage the exchange of messages among a well defined group of nodes by providing communication primitives with different message ordering and reliable delivery semantics.

In terms of message ordering usual options include FIFO, causal, and total order delivery, each one specifying a stronger (and also more expensive to enforce) constraint on how messages are delivered at each node. Total order delivery (which can be causal or not) ensures that if any two nodes in the system receive two messages m_1 and m_2 , both receive either $m1$ before m_2 or m_2 before m_1. Postgres-R took advantage of this property to make sure updates were applied in the same order at all copies by propagating the changes using total order delivery.

In terms of delivery semantics, the exact definitions require more space than the one available here as one needs to consider the notion of *membership to a view* to be precise. For the purposes of this paper, it suffices to consider two main options. Reliable delivery ensures that if a message is delivered to a node that is available, that message will be delivered at all available nodes. Uniform delivery ensures that if a message is delivered to a node, that message will be delivered at all available nodes. The difference between the two is subtle but crucial. Uniform delivery is a much stronger property than reliable delivery because it requires that if a message is delivered to a node, the message is delivered to all working nodes *even if the first node fails*. Postgres-R took advantage of the recovery semantics of database engines and chose reliable delivery rather than uniform delivery (all the implications of this choice were explored in detail to ensure Postgres-R behaved correctly [22]).

The similarities and differences between the approaches to replication in databases and distributed systems were quite interesting and illuminating [35]. Postgres-R was designed in part based on understanding these differences and making the right choice depending on the task: conventional concurrency control within a replica, and group communication to reduce the overhead of enforcing consistency among replicas.

3. POSTGRES-R

3.1 Overview

Postgres-R was built as a collection of nodes containing fully replicated databases. Clients connected to one of the replicas and submitted their transactions to that replica. The corresponding database executed the transaction locally

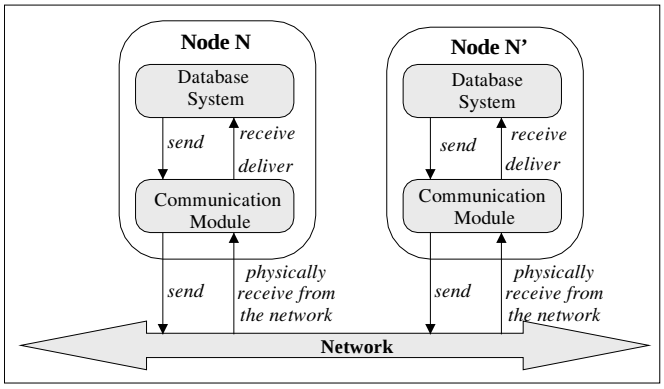

Figure 4: Architecture of Postgres-R

and when it received the commit request from the client, it sent the updates to all the other copies using a group communication service (GCS). Once the proper order for the transactions had been established and validated, the transaction was committed while in parallel all other copies were being updated.

Each Postgres-R node was based on a modified version of PostgreSQL (at that time version 6) plus a communication module implementing the group communication service. The original paper used the group communication system Ensemble [20] for that purpose. As shown in Figure 4, the use of the communication module induces a number of steps when dealing with messages. Postgres-R distinguished between physically receiving a message form the network (the message was received but its total order not established), delivering a message to the database (the communication module forwarded a message with a transaction to the database only when the order was established and the reliability guarantees for the message were met), and receiving a message at the database where it was processed. For reference purposes, Figure 5 illustrates the overall architecture of each node in Postgres-R as an extension of the main components of PostgreSQL.

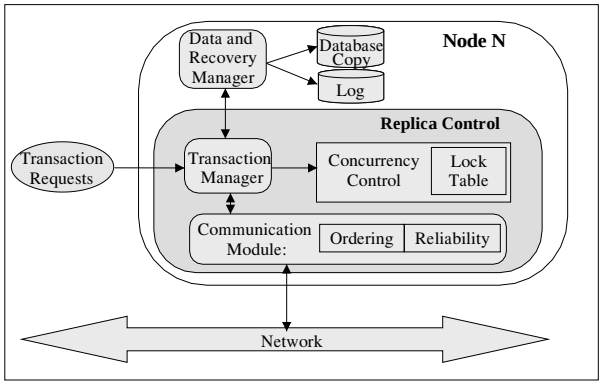

Figure 5: Components of a Postgres-R node

3.2 Transaction Execution

Transaction execution in Postgres-R was very different from conventional database replication solutions. Conceptually, it also used a *read-one-write-all* approach but divided

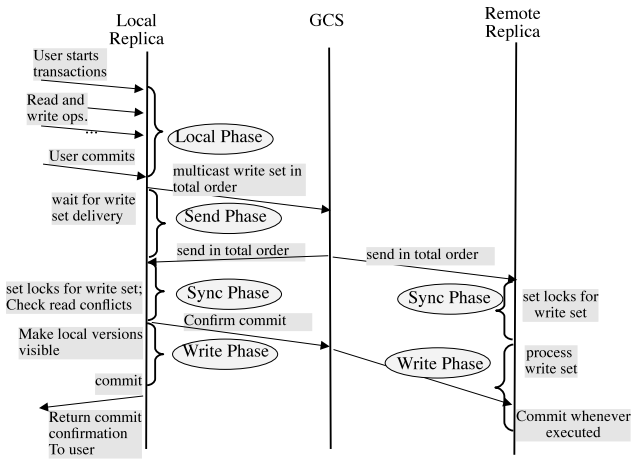

Figure 6: Transaction Execution Phases

transaction execution into 4 different phases: a local phase where a transaction was processed only at one database (called the local replica), a send phase where the updates were propagated, a synchronization phase where the global serialization order was established, and a write phase where remote replicas executed the writes and all replicas committed the transaction (Figure 6).

These 4 phases enabled a number of key features that helped Postgres-R achieve the performance and simplicity that made it so different from traditional approaches.

- **Reduced coordination overhead:** A transaction was first completely executed at a single replica and the write operations only sent at the end of the transaction within a single message. Thus, Postgres-R avoided the coordination overhead of traditional approaches. Until the commit request was submitted the transaction took as long as if there were no replication. Read-only transactions remained completely local without any coordination needed.

- **Local concurrency control for local transactions:** While transactions executed locally, they were isolated from other locally executing transactions by the concurrency control algorithms implemented in the database engine. Thus, existing isolation mechanisms were fully exploited.

- **Total order delivery**: A group communication system was used to propagate the write set to all replicas in total order. This order was used to determine a total serialization order for update transactions across all replicas during the synchronization phase. As all replicas received the write sets in the same order, all replicas could serialize write operations in the same way without any further coordination with other replicas. This simplified global concurrency control significantly and, e.g., eliminated distributed deadlocks entirely.

- **Independent write phases**: To avoid that the response time increased due to having to wait until the transaction was committed at all sites, the local replica would commit the transaction as soon as its position in the total order was known. It had already executed the

write operations during the local phase, and the write phase at the remote replicas was performed independently. As a result, the local replica could confirm the commit to the client without having to wait until the other replicas had executed the transaction.

- **Early lock release**: The write phase at a remote replica was expected to be fast as all write operations were known and no further coordination with other replicas was needed. Thus, locks and resources needed to be kept only for a short time, much shorter than during the local phase at the local replica. Thus, applying the write operations of remote transactions had only little impact on locally executing transactions.
- **Reliable delivery:** Through the use of reliable delivery, even if failures occurred, all available replicas delivered the same set of write sets. With uniform reliable delivery it is even guaranteed that failed replicas delivered a subset of the write sets delivered by the available replicas. In practice, this provided atomicity: all (available) replicas commited the same set of transactions without having to use 2 Phase Commit for each transaction.

As a result of these mechanisms, response times in Postgres-R were barely higher than in a non-replicated system. The only added latency was due to having to send the write set to all copies and to perform the synchronization phase. In our implementation that overhead was very small compared to the time spent in the local phase.

Let's now have a closer look at what happened in each of these phases.

In the *local phase*, all read and write operations of a transaction were executed locally at one replica, namely the replica the client was connected to. During the local phase, concurrency control at the local replica isolated the transaction from other transactions running concurrently at the local replica. Transactions on other replicas were not visible. The version of PostgreSQL we modified used strict two-phase locking, a technical aspect that had important implications in how Postgres-R operated. For the local phase, we simply adopted this concurrency control mechanism and didn't change it. However, updates were performed on shadow copies rather than in-place.

When the client submitted the commit request, a read-only transaction could commit locally. Thus, read-only transactions remained completely local not requiring any coordination with other replicas. For update transactions, a *send phase* used multicast to send all write operations in a single *write set* message to all other replicas and the delivery to the database was done in total order. Even if failures occurred, it was guaranteed that either all or none of the available replicas received the write set, thereby ensuring atomicity.

When the write set arrived at a replica, the *synhronization phase* started. This phase guaranteed that the transaction was properly serialized in regard to all transactions in the system, not only the local ones. An important feature was that each replica performed the synchronization phase locally, with nearly no interaction between the replicas. This was achieved by letting each replica acquire the write locks in the order their write sets were delivered, i.e., the synchronization phases of different transactions were serial. We extended the original locking component of PostgresQL to be able to acquire several locks in a single atomic step. As a result, all replicas ordered write operations in the same way. A tricky issue was that only the local replicas saw the read operations. In Postgres-R we solved this issue by letting the local replica check for read/write conflicts which then would inform the other replicas via an additional commit/abort message whether the transaction could really commit or not. Later versions did not needed to do this as they used snapshot isolation instead (thereby removing read/write conflicts).

If the synchronization succeeded, the *write phase* started. At the local replica, the shadow copies simply became visible copies and the transaction commited. At the remote replicas the write operations were applied and the transaction committed. No further communication took place among replicas. It was also possible for replicas to apply write sets concurrently if the write sets did not conflict.

An additional and very important optimization in Postgres-R was the way write sets were constructed and propagated. Instead of executing the original write operations on the remote replicas as described in traditional eager approaches (or propagating SQL statements as some modern systems do [4, 3, 2]), the write set contained the *after-images* of the affected records. These after images could be applied in a very efficient manner at the remote replicas, significantly reducing the overhead of updating copies. As a result, replicas had more processing power for local transactions, boosting the overall performance of the system.

3.3 Performance of Postgres-R

Postgres-R was able to scale to up to 15 nodes with typical read/write workloads, a major breakthrough at that time. Such performance was possible because Postgres-R exhibited a completely different behavior than systems implementing replication through distributed locking and 2 Phase Commit. As an example, Figure 7 shows the response time of Postgres-R[2].

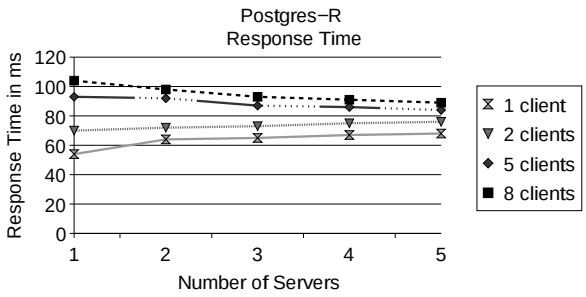

Figure 7: Response time in Postgres-R

With a single server, response times worsened quickly with increasing number of clients, although the overall workload was the same for all tests. This was due to poor connection management in the version of PostgreSQL we used. More interesting in the context of this paper is that with increasing number of servers, the response times did not deteriorate, remaining almost independent of the number of replicas. With higher number of clients, response times

[2]For other performance results, we refer to the original paper or [21]. Note that the experiments for Postgres-R were done on different hardware than the experiments in Figures 1, 2, and 3. The results can only be compared qualitatively.

even slightly decreased as connection management was distributed across several replicas and concurrent transactions did not block or interfered with each other. Such behavior was the basis for achieving scalability while still providing full serializability.

4. LESSONS LEARNED

Postgres-R demonstrated that it was possible to implement consistent data replication in a scalable manner. The key ideas behind its design have survived the test of time and now are used in many systems. In particular, the use of group communication technology, and the execution split into a local phase and a coordination phase have shown to be very efficient. There were, nevertheless, aspects of Postgres-R that, with hindsight, could have been done differently and, in fact, had been changed as technology and the understanding of the problem evolved in the last ten years. In what follows we also point out some of the aspects that did not work that well and were eventually replaced for other solutions.

4.1 Architecture

Kernel vs. middleware replication. One of the key aspects in which Postgres-R differs from most of the later developments was that it was implemented inside a database system. While this provided a great opportunity for optimizations and allowed for a tight coupling of concurrency control and replica control, it was an invasive approach and heavily dependent on a particular database implementation[3]. Many of the later systems chose to implement replication outside the database in a middleware layer. A middleware layer has the big advantage that the underlying database does not need to be modified (from an engineering point of view, modifying PostgreSQL was the most involved aspect of Postgres-R). It also leads to a nice separation of concerns, enabling heterogeneous environments, and allowing the use of database systems whose source code is not available. In [10], a detailed analysis of middleware-based approaches is provided.

Many of these middleware-based approaches use group communication to coordinate transactions at the global level and to guarantee message delivery at all replicas. For instance, Middle-R [33] was a relatively straightforward extension of Posgres-R with similar functionality and behavior but without modifying the database. C-JDBC [11] extended the JDBC layer to turn the databases into RAID like storage systems. Ganymed showed that middleware replication can support several hundred databases in a multi-tenant setting, a configuration that has gained in relevance in cloud computing scenarios [37]. Vandiver et al. have used middleware based replication to implement a replicated set of databases (homogeneous or heterogeneous) that tolerates byzantine failures [41].

Concurrency Control. A second development was a more extensive use of different levels of consistency and multi-version systems, especially the use of snapshot isolation as the global isolation level. While lower levels of consistency were already explored relatively early [22], Posgres-R was based on locking, as that was the concurrency control mechanism available in PostgreSQL at that time. The challenge was the handling of read locks which was cumbersome. Middleware-based approaches that used locking had problems to implement concurrency at a fine granularity [11, 33]. In snapshot isolation, read operations are performed on a snapshot of the data and conflicts are only between write operations. The replica control mechanism can thus ignore read operations and leave their coordination to the database system, even in middleware-based approaches. Ganymed [36], for instance, used a primary copy approach where the updates were all executed on a primary replica that implements snapshot isolation while the other replicas are read-only. There exist several approaches that exploit snapshot isolation in a multi-master environment, relying on group communication primitives for ordering write operations and enabling fine-granularity concurrency. Examples are the work by Lin et al. [28], Elnikety et al. [15, 14] or Muñoz-Escoí et al. [31]. A later version of Postgres-R [43] is also based on snapshot isolation, taking advantage of it becoming available in PostgreSQL. The result were highly reduced abort rates compared to the original Postgres-R.

4.2 Overview of commercial systems

The advantages offered by architectures like the one of Postgres-R can be exploited in a variety of ways. This has lead to a wide range of systems implementing variations of the same basic ideas, targeting different scenarios.

Continuent's *Tungsten* is a partially open source platform based on a group communication middleware layer that propagates changes to all copies in a primary copy configuration (master/salve) [3]. The reason for this architecture is, according to Continuent's literature, that primary copy scales better and can be run in a larger number of nodes than multi-master replication. In Tungsten, SQL statements can be rewritten to theoretically support heterogeneous databases (Oracle and MySQL) acting as replicas of each other, an idea that has also been explored in the Ganymed system [38] using Oracle, DB2, and PostgreSQL.

MySQL/Galera [2] is an open source system that implements a multi-master approach over MySQL where all updates are propagated to all copies. The approach is in clear contrast to that of Continuent as MySQL/Galera argues that multi-master offers better scalability and performance behavior than a primary copy architecture. This has also been argued in the research literature [13] although no performance comparisons of existing systems have been done. The issue is quite relevant in practice and deserves more attention, although it is likely that each configuration is suitable to different workloads: multi-master probably fits better OLTP scenarios while primary-copy is likely to be more adequate for business intelligence and real-time OLAP workloads.

Xkoto's *Gridscale* implements fully replicated databases with SQL forwarding of all changes to each replica using a synchronization protocol similar to group communication [4]. Unlike Postgres-R, updates are not forwarded after having been executed in one database. Instead, the SQL statements that modify the database are forwarded to all copies ensuring total order delivery. The statements are changed in flight to deal with non-deterministic operators such as time of day or random numbers, thereby ensuring that all

[3]The PostgreSQL community continues to work on Postgres-R [1], but keeping it up-to-date with the evolving main database system is difficult.

copies are identical. Gridscale replies to the client as soon as one of the underlying databases reply to avoid having to wait until all copies are updated. This mechanism is conceptually similar to the one used in Postgres-R, although in Postgres-R transactions are submitted to a database while in Gridscale transactions are submitted to the replicated system as a whole.

Recently, Microsoft has launched a number of initiatives around SQL Azure wher a highly optimized agreement protocol is used to synchronize share nothing SQL Servers in a cloud setting [9]. The application scenario is cloud computing and the goal of replication in this case is to increase teh elasticity of the database. An interesting aspect of SQL Azure is the inclusion of *consistency domains* to adjust the degree of consistency provided as the system scales.

4.3 Further Related Work

Because of the involved interplay between the semantics of group communication and transactional semantics, there has been a significant amount of research exploring the problem in more detail, particularly the theoretical and correctness aspects. For reasons of space, we cannot cover all this work in any detail. Related topics that have been addressed in the literature include, e.g.: state machine approaches [34], wide area replication [5, 27], the use of total order in lazy replication [32], analysis of correctness criteria [12, 26], and the reliability and performance of database replication based on group communication [40]. New protocols for enforcing consistency are also starting to appear, e.g., [25], an area where much more work is still needed.

5. CONCLUSIONS

Ten years ago, Postgres-R showed that it was possible to implement consistent database replication without a significant performance loss and in a scalable manner. It did so by combining results from distributed systems with results from databases in a unique architecture that helped solved many of the problems encountered by conventional solutions to database replication. Nowadays, the main ideas behind Postgres-R have become established, being used in numerous commercial systems and research projects.

Beyond the solution it provided to database replication, the key lesson of Postgres-R is the advantages and new perspectives to be gained by looking at problems in one research area with the expertise and tools of another research area. As we enter a new era dominated by cloud computing, software and hardware appliances, and remote services, this is a lesson worth keeping in mind as today's problems in system design can no longer be solved by looking at them from the confines of single and isolated research communities.

Acknowledgments

Many people have contributed to the initial work and its development over the last decade. The research in the original paper was partially funded by ETH Zurich under the DRAGON project (Nr. 41-2642.5), which started when the two authors were working in the group of Hans-Joerg Schek. Andre Schiper and Fernando Pedone from EPF Lausanne were our partners in the DRAGON project and a considerable help with all concepts related to group communication. Marta Patiño-Martínez and Ricardo Jiménez-Peris worked as post-docs at ETHZ in extending the ideas of Postgres-R to a middleware platform, Middle-R. Development of Postgres-R continued at McGill University for several years, and many master students both at ETHZ and McGill contributed code and ideas: Win Bausch, Michael Baumer, Ignaz Bachmann, Mabrouk Chouk, Shuqing Wu, and WeiBin Liang. At ETHZ, follow-up work with Christian Plattner lead to the *Ganymed* system. More recently, Tudor Salomie and Ionut Subasu have applied these concepts in multicore architectures as part of the *Multimed* project; Philipp Unterbrunner has developed e-cast, an agreement protocol tailored for partial replication. At McGill, Yi Lin extended some of the ideas of Postgres-R to middleware platforms, with a focus on new consistency criteria and wide-area replication. Finally, the PostgreSQL community has provided us with a lot of feedback on the various versions of Postgres-R; their comments and wish-lists have been a constant motivation for our research.

6. REFERENCES

[1] *http://wiki.postgresql.org/wiki/Postgres-R.*
[2] *http://www.codership.com/products/mysql_galera.*
[3] *http://www.continuent.com/.*
[4] *http://www.xkoto.com/.*
[5] Y. Amir and C. Tutu. From total order to database replication. In *ICDCS*, 2004.
[6] H. Attiya and J. Welch. *Distributed Computing.* Wiley-Interscience, 2004.
[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
[8] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC*, page 7, 2000.
[9] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In *SIGMOD*, pages 1021–1024, 2010.
[10] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*, pages 739–752, 2008.
[11] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX*, pages 9–18, 2004.
[12] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726, 2006.
[13] S. Elnikety, S. G. Dropsho, E. Cecchet, and W. Zwaenepoel. Predicting replicated database scalability from standalone database profiling. In *EuroSys*, pages 303–316, 2009.
[14] S. Elnikety, S. G. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, pages 117–130, 2006.
[15] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS*, pages 73–84, 2005.
[16] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *PODS*, pages 1–7, 1983.
[17] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, pages 173–182, 1996.

[18] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1), 2006.

[19] R. Guerraoui, R. C. Oliveira, and A. Schiper. Atomic updates of replicated data. In *EDCC*, pages 365–382, 1996.

[20] M. Hayden. The ensemble system. Technical report, Cornell University, TR98-1662, 1998.

[21] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, ETH Zürich, Departement of Computer Science, 2000. Diss. ETH No. 13864.

[22] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *ICDCS*, pages 156–163, 1998.

[23] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, pages 134–143, 2000.

[24] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[25] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC*, pages 312–313, 2009.

[26] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. E. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2), 2009.

[27] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Consistent data replication: Is it feasible in wans? In *Euro-Par*, pages 633–643, 2005.

[28] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*, pages 419–430, 2005.

[29] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[30] S. Mullender. *Distributed Systems*. Addison-Wesley, 1993.

[31] F. D. Muñoz-Escoí, J. Pla-Civera, M. I. Ruiz-Fuertes, L. Irún-Briz, H. Decker, J. E. Armendáriz-Iñigo, and J. R. G. de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, pages 401–410, 2006.

[32] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3):237–267, 2001.

[33] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[34] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[35] F. Pedone, M. Wiesmann, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474, 2000.

[36] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, pages 155–174, 2004.

[37] C. Plattner, G. Alonso, and M. T. Özsu. Dbfarm: A scalable cluster for multiple databases. In *Middleware*, pages 180–200, 2006.

[38] C. Plattner, G. Alonso, and M. T. Özsu. Extending dbmss with satellite databases. *VLDB J.*, 17(4):657–682, 2008.

[39] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Commun. ACM*, 39(4):84–87, 1996.

[40] A. Sousa, J. Pereira, L. Soares, A. C. Jr., L. Rocha, R. Oliveira, and F. Moura. Testing the Dependability and Performance of Group Communication Based Database Replication Protocols. In *DSN*, 2005.

[41] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, pages 59–72, 2007.

[42] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[43] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433, 2005.