Database Replication Techniques: a Three Parameter Classification*

Matthias Wiesmann⋆ Fernando Pedone† André Schiper⋆
Bettina Kemme‡ Gustavo Alonso‡

⋆Département de Systèmes de Communication
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne, Switzerland

†Software Technology Laboratory
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA

‡Institute of Information Systems
Swiss Federal Institute of Technology in Zürich
CH-8092 Zürich , Switzerland
E-mail: dragon@lsemail.epfl.ch

Abstract

Data replication is an increasingly important topic as databases are more and more deployed over clusters of workstations. One of the challenges in database replication is to introduce replication without severely affecting performance. Because of this difficulty, current database products use lazy replication, which is very efficient but can compromise consistency. As an alternative, eager replication guarantees consistency but most existing protocols have a prohibitive cost. In order to clarify the current state of the art and open up new avenues for research, this paper analyses existing eager techniques using three key parameters. In our analysis, we distinguish eight classes of eager replication protocols and, for each category, discuss its requirements, capabilities, and cost. The contribution lies in showing when eager replication is feasible and in spelling out the different aspects a database replication protocol must account for.

1. Introduction

In the distributed systems community, software based replication is seen as a cost effective way to increase availability. In the database community, however, replication is used for both performance and fault-tolerant purposes thereby introducing a constant trade-off between consistency and efficiency. In fact, many commercial [16, 26] and research databases [35] are based on the asynchronous replication model (also called *lazy update model*) where changes introduced by a transaction are propagated to other sites only after the transaction has committed. This results in minimal overhead but inconsistencies among the copies may arise. This characteristic trade-off does not necessarily imply that consistency is not important in databases. It is well known to users and designers that the inconsistencies created by lazy replication techniques can be very difficult to solve. It is also well known that such inconsistencies can be eliminated by using synchronous replication models (also called *eager* replication, i.e., a transaction synchronises with all copies before it commits). Unfortunately, it is by no means trivial to design efficient eager replication protocols. In practice, given the serious limitations of traditional data replication techniques (deadlocks, overhead, lack of scalability, unrealistic assumptions), many database designers do not regard eager replication as a feasible option [13].

It has been only recently that efficient eager replication protocols have started to appear. Most of these new protocols are based on group communication primitives and the results obtained so far seem to indicate that this approach can solve most of the problems associated with eager data replication. These efforts are the main context for our work. In the last few years, as part of the DRAGON project [17], we have focused on enhancing database replication mechanisms by taking advantage of some of the properties of group communication primitives. We have shown how group communication can be embedded into a database [1, 28, 29] and used as part of the transaction manager to guarantee serialisable execution of transactions over replicated data [19, 18]. We have also shown how some of the overhead associated with group communication can be hidden behind the cost of executing transactions, thereby greatly enhancing performance [20]. These results prove the importance of finding synergies between distributed systems and database replication techniques and the need for a common understanding of the models used by the two communities in order to address complex research issues.

As part of this work, we have systematically explored the space of eager database replication strategies [38]. In this paper, we present the results obtained in an attempt to further clarify the spectrum of possible eager replication protocols and to point out new directions for research. One of the novel aspects of our classification schema is that it integrates protocols from both databases and distributed systems. Existing taxonomies of data replication techniques take into account a broad spectrum of replication schemes, including many with weak consistency and availability properties, but either without including techniques based on group communication [7],

*Research supported by EPFL-ETHZ DRAGON project

This paper appeared in the Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000), pages 206-215, Nürnberg, Germany, October 2000. IEEE Computer Society.

or considering only simple cases [8]. Our classification is much more concise than existing attempts and emphasises the synergy between communication and transaction management. The benefits of our classification effort are numerous. First, it has allowed us to identify the key components of a database replication protocol. Second, it has led us to better understand the role played by each component and its influence on the nature of the protocol. Some of the strategies we consider have been described in the literature, but not all. Third, the classification forms the basis for quantitative comparison of the various replication strategies identified, an endeavour in which we are currently engaged. These quantitative comparison will shed light on many different aspects of eager replication and the role that transaction management and group communication play in implementing eager replication protocols.

The rest of the paper is structured as follows. Section 2 describes our replicated database architecture and the system model. Section 3 introduces the classification parameters. Section 4 discusses the various replication techniques obtained by combining these parameters. Section 5 describes the role and impact of each parameter and concludes the paper.

2. System Model and Architecture

We consider a set of database clients, and a set $S = \{s_1, s_2, ..., s_n\}$ of database servers (see Figure 1). The database is fully replicated on every server s_i, i.e., every server contains a copy of the whole database. A client connects to one of the database servers, say s_i, to execute a transaction. Transactions are sequences of read and/or write operations followed by a commit or abort operation. Transactions that contain only read operations are called *queries*, and transactions that contain read and write requests are called *update transactions*. Once a client is connected to a database server s_i, it sends the operations of the transaction to s_i for execution. The transaction can be submitted either operation by operation, or in a single message. In the former case, called *interactive transaction*, after submitting an operation, the client waits for an answer from the server (e.g., the results of a read request), after which it sends the next operation. Sending the transaction as a single message is called a *service request* [6], that is, a call to a procedure stored on the database servers.

Once the transaction is completed, the server s_i sends the transaction outcome to the client and the connection between the client and s_i is closed. If the transaction was submitted operation by operation, the outcome is a commit or abort confirmation. In case of a service request, the transaction outcome also includes the results of the request. If s_i fails during the execution of the transaction, the transaction aborts. In this case, it is up to the client to retry the execution, either by connecting to a different database server s_j, or to the same server s_i later (after s_i recovers).

The correctness criterion for transactions that we consider is one copy serializability [2]. It ensures that any interleaved execution of transactions is equivalent to a serial execution of these transactions on a single copy of the database. Furthermore, we concentrate on *eager* replication techniques. Using eager replication, updates are propagated to the replicas within the boundaries of a transactions, and hence, conflicts are detected before the transaction commits. This approach provides data consistency in a straightforward way.

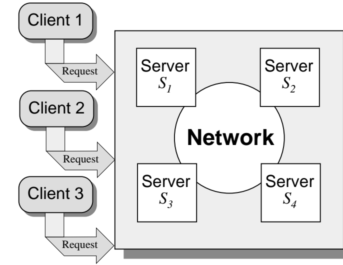

Figure 1. System architecture

Database servers communicate among each other via point-to-point or 1-to-n communication. The latter is provided by group communication systems. We consider group communication primitives that differ in two parameters: reliability of message delivery and ordering. Regarding delivery guarantees, *reliable broadcast* ensures that a message sent by a correct database server, or delivered by a correct database server, is eventually delivered to all correct database servers. *Uniform reliable broadcast* ensures that a message delivered by any server (i.e., it can be correct or fail directly after delivering the message) is eventually delivered to all correct database servers. Regarding ordering semantics, there might be no ordering requirements (messages might be delivered in arbitrary orders at the different sites) or a total order delivery of messages, i.e., if two database servers deliver both messages m and m' then they deliver them in the same order. This semantics is provided by the so-called *total order broadcast*, also called *atomic broadcast*.

3. Classification Criteria

Eager replication protocols can be organised according to three parameters that determine the nature and properties of the protocol (and, in some cases, also its performance). These parameters are: the server architecture (primary copy or update everywhere), how changes or operations are propagated across servers (on a per operation or a per transaction basis), and the transaction termination protocol (voting or non voting).

3.1. Server Architecture

The first key parameter to consider is where transactions are executed in the first place. Gray et al. [14] have identified two possibilities:

Primary copy replication requires to have a specific site –the primary copy– associated with each data item. Any update to the data item must be first sent to the primary copy where it is processed (executed or at least analysed to establish its serialisation order). The primary copy than propagates the update (or its results) to all other sites. In [38] it was shown that primary copy approaches closely resemble what in distributed systems is known as passive replication [15]. As expected, primary copy approaches introduce a single point of failure and a bottleneck. These limitations can be solved by making the protocol more complicated. Thus, if the primary crashes,

one of the other servers takes over the role of primary which requires an election protocol. Similarly, to avoid bottlenecks, databases do not make a single site the primary for all data items. Instead, the data is partitioned and different sites become the primary for different data subsets. In what follows, we will mostly ignore these aspects of the protocols since they are orthogonal to the discussion in the paper.

Update everywhere replication allows updates to a data item to be performed anywhere in the system. That is, updates can concurrently arrive at two different copies of the same data item (which cannot happen with primary copy). Because of this property, update everywhere approaches are more graceful when dealing with failures since no election protocol is necessary to continue processing. Similarly, in principle, update everywhere introduces no performance bottlenecks. However, update everywhere may require that instead of one site doing the work (the primary copy) all sites do the same work. If one is not careful with the design, update everywhere may affect performance much more than primary copy approaches.

3.2. Server Interaction

The second parameter to consider involves the degree of communication among database servers during the execution of a transaction. This determines the amount of network traffic generated by the replication algorithm and the overall overhead of processing transactions. This parameter is expressed as a function of the number of messages necessary to handle the operations of a transaction (but not its termination). Moreover, the type of primitive used to exchange these messages will also play a role in determining the properties of the protocol from a serialisation point of view. We consider two cases:

Constant interaction, which corresponds to techniques where a constant number of messages is used to synchronise the servers for a given transaction, independently of the number of operations in the transaction. Typically, protocols in this category exchange a single message per transaction by grouping all operations of the transaction in a single message.

Linear interaction, which typically corresponds to techniques where a database server propagates each operation of a transaction on a per operation basis. The operations can be sent either as SQL statements or as log records containing the results of having executed the operation in a particular server.

3.3. Transaction Termination

The last parameter to consider is the way transactions terminate, that is, how atomicity is guaranteed. We distinguish two cases:

Voting termination requires an extra round of messages to coordinate the different replicas. This round can be as complex as an *atomic commitment protocol* (e.g., the *two-phase commitment protocol* (2PC) [2]), or as simple as a single confirmation message sent by a given site.

Non-voting termination implies that sites can decide on their own whether to commit or abort a transaction. Non-voting techniques require replicas to behave deterministically. This, however, is not as restrictive as it may appear at first glance since the determinism only affects transactions that are serialised with respect to each other. Transactions or operations that do not conflict can be executed in different orders at different sites. Many of the issues related to determinism in databases when communication primitives are used have been studied in detail in [1] and are beyond the scope of this paper.

4. A Plethora of Replication Techniques

In this section, we explore all the combinations that result from the classification parameters in Section 3. In each case, the general framework of replication techniques that fit the combination of parameters is described. Existing replication techniques matching this combination are also listed. For each combination, the requirements needed to build a replication technique fulfilling the classification criteria are given. Those requirements are expressed either for the communication infrastructure, or the database system on each server. Requirements on the communication system are usually ordering or uniformity constraints on the delivery of messages to database servers. For the database the requirement is determinism.

Server Interaction	Transaction Termination	Server Architecture: Update Everywhere	Server Architecture: Primary Backup
Constant	Voting	§ 4.1.2	§ 4.2.2
Constant	Non Voting	§ 4.1.1	§ 4.2.1
Linear	Non Voting	§ 4.1.3	§ 4.2.3
Linear	Voting	§ 4.1.4	§ 4.2.4

Figure 2. The different techniques

Figure 2 summarises the different classification parameters. Each entry in the table represent one possible combination of parameters. For each combination, the section discussing the combination is indicated.

Determinism Point. One important notion while describing non-voting replication techniques is determinism. Because the different replica cannot communicate to check if they all reached the same serial order, they need to behave in a deterministic way from some point in time to ensure one-copy serialisability. We define the *determinism point* as the point in the execution of a transaction after which the processing will be deterministic. Formally, the determinism point dp of transaction t, is the first operation of transaction t such as any operation o after dp executes in a deterministic way. This means that once dp is executed, the execution of the rest of transaction t is deterministic. The main implication is that once this point is reached the position of the transaction in the serial history can be determined.

The notion of a determinism point is related to the notion of a serialisation point [6]. Serialisation points (sp) are used to enforce a strict ordering upon a database system. By definition, if the sp_1 of transaction t_1 executes before sp_2 of transaction t_2, then t_1 is before t_2 in the serial history. In other words, once the serialisation point of a given transaction t has been executed, then the position of transaction t in the serial history is known and fixed. Therefore serialisation point are also determinism points. The reverse is not true.

Note that the determinism point is a property given by the local database system of each replica: depending on the form of the transaction accepted by the server and the type of concurrency control, the database server can have different determinism points, for instance at the beginning of the transaction ($dp = begin$), or at the end ($dp = commit$ or $dp = abort$), etc.

4.1. Update Everywhere

In update everywhere techniques, the clients can send their requests to any server. The server contacted will act as the *delegate* for the requests submitted by the client. The delegate will process the requests and synchronise with the other servers to ensure one copy serializability.

4.1.1 Update Everywhere – Constant Interaction – Non-Voting Techniques

Description Figure 3 shows the basic structure of such a replication technique. In the discussion that follows, we assume that there is only one network interaction between servers. This simplification makes the description clearer and does not leave out any important detail. The protocols in this category execute according to the following steps:

① The transaction starts on the delegate server.
② The transaction is processed in a non-deterministic way.
③ The *determinism point* is reached.
④ The transaction is sent to all servers using an atomic broadcast primitive.
⑤ Processing continues on all replicas in a deterministic way.
⑥ Each replica terminates the transaction in the same way.

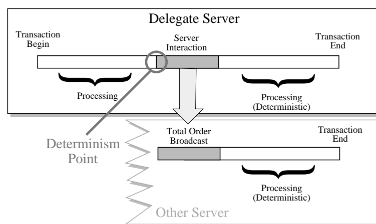

Figure 3. Update everywhere, constant interaction, non-voting

References. Previous works following the Update Everywhere – Constant Interaction – Non-Voting model can be divided according to where the determinism point is placed.

If the *point of determinism* is at the beginning of the transaction, the whole transaction processing is deterministic, and the role of the delegate server is simply to forward the transaction using the total order broadcast primitive (step 2 in the description above does not really apply). The delegate simply acts as a *proxy* for the client, contacting all servers to process the client's request. This approach closely correlates with *active replication* [33]. An early example of such an approach can be found in [31]. The technique described in [20] also uses total order broadcast and an early determinism point extended by certain forms of optimistic transaction execution.

Techniques with the *determinism point* at the end of the transaction processing are called certification techniques [29, 19, 18]. In this case, the whole transaction is handled in a non-deterministic way by the delegate, and only the last stage of the processing, the *certification*, is deterministic. This certification stage is executed at all sites and decides if a transaction will be committed or aborted. In [29] information about both read and write operations is sent to all sites in order to detect conflicts during the certification phase. [19, 18] use snapshot isolation instead of serializability to avoid conflicts between read and write operations, and hence, the certification phase is restricted to write operations. In all cases, the certification phase is deterministic.

Requirements. We discuss correctness by distinguishing transaction isolation (one copy serializability) from transaction atomicity (i.e., all or none of the databases commit a transaction).

Independently of where the determinism point lies, the mechanism used to guarantee one copy serializability is always the same. The total order used in the broadcast acts as a guideline to all sites. Each site guarantees that its local serialisation order will follow the total order, and thus all sites will produce the same serialisation order (since they see the same total order). The differences in the protocols lie on the determinism point. For protocols that place the determinism point at the beginning of the transaction, the total order suffices. For protocols that place the determinism point at the end of the transaction, things are a bit more complicated. In particular, when confronted with situations where a transaction needs to be aborted, a delegate server can only abort transactions not yet seen by other sites. The protocol must ensure that as soon as transactions are seen by other sites, there will be no problems with their scheduling or that all sites will end up aborting the transaction.

Related to this, techniques in this category do not need a distributed deadlock detection system. Since transactions are sent in one step using a total order broadcast, locks for the whole transaction can be acquired atomically and in the same order at all sites thereby preventing deadlocks.

Transaction atomicity is enforced by a uniform reliable broadcast of the messages, and the deterministic behavior of the different servers. This guarantees that whenever a server delivers a message and commits the transaction, each server will deliver the message (uniformity) and commit the transaction (determinism).

Discussion. A determinism point at the beginning of the transactions usually implies that the datasets of the transactions are known in advance, e.g., stored procedures. Thus, what it is being sent to all sites is the invocation of the procedure. It follows that all sites must execute this procedure deterministically since there is no voting phase. This is one of the limitations of these approaches since making all sites completely deterministic can be difficult in practice.

Using determinism points at the end of the transactions might be more feasible. It does not require to know the transactions in advance since it can be implemented by deferring writes to the end of the transactions or by executing the transaction on a shadow copy. Nevertheless, the fact that the determinism point comes at the end of the transaction has several implications. The main one is that there is a degree of optimism in the execution. Servers accept many transactions but

might abort some of them. There is a tradeoff between early determinism points and abort rate. Having the determinism point early means low aborts due to conflicts. Having this point late implies having higher chances of conflicts. Intermediate solutions, where the determinism point is in the middle of the transaction would be a compromise. One way of implementing such a solution would be to execute all the reads in the beginning of the transaction, and then do all the writes in an atomic and deterministic step. In this case, the determinism point would be the first write operation.

4.1.2 Update Everywhere – Constant Interaction – Voting Techniques

Description. Figure 4 shows the basic structure of a replication technique in this category. This technique is similar to the one in the previous section, in that all the interactions are done using one communication phase. Additionally, a final voting phase is executed at the end of the transaction's execution to ensure that all replicas agree on the outcome. The execution is done in the following way:

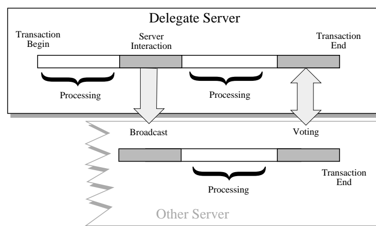

Figure 4. Update everywhere replication, constant interaction, voting

① The transaction starts on the delegate server.
② The transaction is processed in a non-deterministic way.
③ The transaction is broadcast to all servers.
④ Processing continues on all replicas.
⑤ A voting termination phase takes place.
⑥ Each replica terminates the transaction according to the voting protocol.

References. As an example of this technique, in [11] the delegate server broadcasts a transaction to the other sites immediately when it is submitted and a total order broadcast is used (the total order being derived using synchronised clocks). Also here, the total order is used as a guideline at every site to serialise transactions. The final voting phase is only used to ensure atomicity in the case of different types of failures. Because these failures can occur at any site, a 2PC protocol is needed.

Another example of constant interaction with voting is the serializability based protocol presented in [19, 18]. In this protocol, the transaction is locally executed at the delegate site and then sent to the other sites using a total order broadcast primitive. Here, the delegate site is the only one to decide whether the transaction can commit or must abort. Because the situation leading to an abort (due to serialisation problems of local reads and global writes) is not seen by all sites, the delegate site needs to communicate the decision to all other sites. This means that the voting is not a 2PC protocol, but a single message that indicates whether the delegate server has committed or aborted the transaction. As a consequence, while the delegate site has the choice to commit or abort the transaction unilaterally, the other sites must behave deterministically in the sense that they have to obey the commit/abort decision of the delegate.

A third example is the optimised form of 2PC described in [2]. In this protocol, write operations are deferred to the end of the transaction, and the first phase of 2PC (vote request) also contains the transaction updates. Participants in the protocol respond with a *yes* vote if they can obtain the locks for those updates. Otherwise they respond *no* and the transaction is aborted. In this case, the only interaction is the enhanced version of the 2PC protocol.

Requirements. In principle, protocols of this type could use any form of broadcast primitive. However, the type of broadcast primitive used determines the voting phase. If the primitive cannot guarantee that all sites will do the same, then the voting phase can only be 2PC and, as part of this phase, discrepancies among sites must be resolved. Furthermore, distributed deadlocks might occur and must be resolved. If the broadcast is totally ordered, then the requirements are similar to those of Section 4.1.1 (e.g., also no need for global deadlock detection).

How atomicity is guaranteed depends on the protocol. If 2PC is used, then it guarantees the atomicity. Otherwise, the primitive used for broadcasting the transaction must be uniform.

Discussion. Having a voting phase relaxes the determinism requirements on the database servers. In practice, and given that complete determinism in a database server is difficult to achieve, many protocols include a voting phase in one form or another. Nevertheless, some limited form of determinism and the use of a total order broadcast considerably simplifies the protocols.

While a total order broadcast will still help to decrease conflicts, having a voting phase consisting of a complete atomic commitment allows to relax the requirements on the total order, e.g., uniformity [39]. Also weaker forms of total order can be used, e.g., a *most of the time* total order: the total order condition is maintained most of the time, but sometimes *performance failures* occur and out of order delivery might happen [9]. Such performance failures would be detected and corrected at commit time, taking advantage of the voting phase.

4.1.3 Update everywhere – Linear Interaction – Non-Voting Techniques

Description. This category is somewhat misleading. Non-voting implies that there is no round where the fate of the transactions can be agreed upon. Therefore, these protocols must be fully deterministic. Sending operations one at a time requires that all sites treat them in exactly the same way. Nevertheless, at the end the delegate site has to indicate that the transaction has finished. This implies that there is a *termination* message. Assuming this termination message is not used for voting, the general structure of techniques in this category are outlined in Figure 5:

① The transaction starts on the delegate server.
② The first operation is sent to all servers using an atomic broadcast.

③ The first operation is executed on all servers.
④ Items (2) and (3) are repeated until the transaction ends.
⑤ The delegate sends a termination message to indicate the end of the transaction.

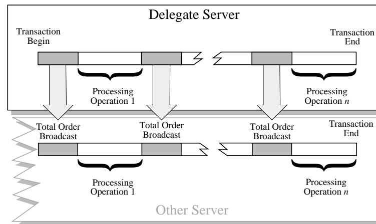

Figure 5. Update everywhere, linear interaction, non-voting

References. An example of this techniques is presented in [1]. Each operation (reads included) is broadcast (uniform total order) to all sites, and sites must behave deterministically in order to react identically to each operation. Techniques in this category are very similar to replicated persistent objects [23].

Requirements. Since there is no voting phase, atomicity can only be guaranteed by sending operations using uniform reliable broadcast. 1-copy-serializability is the result of the local concurrency control mechanism used at each site, and the determinism across sites. Because sites are fully deterministic, deadlocks must be assumed to be resolvable in a deterministic fashion [1].

Discussion. This technique has the major drawback of requiring absolute determinism on all sites, which is a very strong requirement. In addition, there is considerable network overhead since each operation results in a totally ordered broadcast. In general, this technique has not been pursued in the literature as a viable option.

4.1.4 Update everywhere – Linear Interaction – Voting Techniques

Description. This form of database replication technique is the most studied in the literature. Among its many variations, one of the best known is the read-one-write-all technique [2]. Figure 6 shows the interactions of techniques in this category:

① The transaction starts on the delegate server.
② Each operation is broadcast to a quorum of sites.
③ Each operation is executed on its quorum.
④ Items (2) and (3) are repeated until the transaction ends.
⑤ A voting protocol is executed.
⑥ Each replica terminates the transaction according to the voting protocol.

References. This category includes all the traditional database replication protocols: read-one/write-all, write-all-available, and quorums [2]. Most of the effort in this area has been devoted to provide different ways to build quorums. Good surveys of early solutions are [10, 2]. Later approaches mainly optimise quorum sizes and communication costs or analyse the trade-off between quorum sizes and fault-tolerance [21, 32, 36]. In [34] multicast primitives with different ordering semantics are used. The authors propose algorithms using reliable multicast or causal multicast which

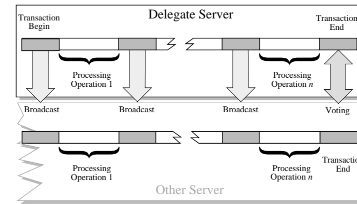

Figure 6. Update everywhere, linear interaction, voting

require an atomic commitment protocol to guarantee serializability.

Requirements. 1-copy-equivalence is achieved by executing each read operation on a read quorum of replicas, each write operation on a write quorum. With this, each site follows a local concurrency control protocol that guarantees serializability, typically 2-phase-locking [37] or timestamp based algorithms [3]. Atomicity is guaranteed by the 2PC protocol during the voting phase.

Discussion. This technique is very well understood. However, in spite of the amount of work invested in this technique, it is not very relevant in practice [13]. The reason is that it has a high overhead (because of the linear number of messages) and has proven to significantly limit scalability due to deadlocks. In addition, when the voting phase involves a complete atomic commitment, the client only gets the response once all replica have committed the changes: this can result in very long response times.

4.2. Primary Copy

In primary copy techniques, the clients must send their requests to one particular server. This server is the *primary*. Because there is only one server executing the transactions, there are no conflicts across the servers. The only thing that has to be assured is that there is only one primary in the system at any time. As this problem is orthogonal to the general architecture, it will not be discussed here. The primary copy approach is widely used in databases to minimise conflicts among transactions executed over replicated data. From now on we will refer to the sites that are not the primary copy for a data item d as backups of d. The backups only install the changes sent by the primary. A site can be the primary for a subset of the data, and the backup for the other data. Such a site is called an *active* backup. A site that is not the primary for any data is called a *passive* backup.

Backup and replication in databases. Database primary-copy techniques are classified along two dimensions: atomicity of transactions (*1-safe* and *2-safe*) and recovery time (*hot-* and *cold*-standby) [14].

An example of hot-standby is Tandem's Remote Data Facility [5, 24, 25] where a server, the *primary*, uses a single backup computer (the *secondary* or backup). Under normal operation, a client sends requests to the primary, and the log records generated at the primary are sent to the backup and

immediately applied. Thus, the backup is an exact replica of the primary, which allows the backup to take over almost immediately upon failure of the primary. If the secondary or backup does not immediately install the changes, then it will need to do so when the primary fails. Because this takes time, there is some delay between the time the primary fails and the time the backup can take over. For this reason, if the backup does not apply the changes as they arrive, the mechanism is called cold-standby.

To achieve consistency, the normal mode of operation is as follows: a transaction takes place in the primary, log records are sent to the secondary, a transaction applies those log records in the secondary, and a 2PC protocol is used to guarantee the combined atomicity of both transactions. Note that since there are only two participants involved, the primary and the backup, some optimisations of 2PC can be implemented [22, 14]. When using 2PC, this approach is known as *2-safe* and it is similar in some aspects to the very-safe case of [14]. Contrary to the 2-safe policy, the *1-safe* policy does not require the primary to wait for the secondary. It commits its transaction independently. There is, of course, the risk of data loss when the backup takes over but in practice the 1-safe policy is often preferred over the 2-safe policy due to its lower overhead. Algorithms for the maintenance of remote copies under the 1-safe and 2-safe policies are discussed in [12] and [4] respectively.

4.2.1 Primary Copy – Constant Interaction – Non-Voting

Description. Techniques in this category generally correspond to cold-standby scenarios. The protocols have the following general outline (see Figure 7):

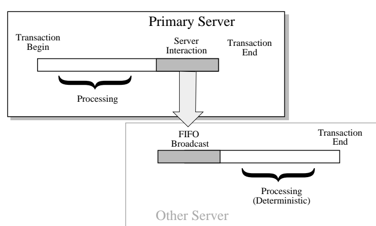

Figure 7. Primary copy, constant interaction, non-voting

① The transaction is executed at the primary.
② When the transaction terminates, the corresponding log records are sent to all backups.
③ The primary commits the transaction without waiting for the backups to install the changes.
④ The backups eventually install the changes.

The concrete nature of the protocol depends on the type of broadcast primitive used. In its simplest form, the protocol is based on FIFO delivery, in order to ensure that transaction changes are installed at the backup in the same order they were executed at the primary.

Requirements. In the case of passive backups, as long as the transaction changes are installed in the same order as in the primary, the backups will consistently reflect what has happened at the primary. Thus, if the primary sends changes in FIFO order and is producing correct histories, so do the backups. This also holds in the case of active backups as long as transactions only access data for which the executing site is the primary. Care has to be taken if transactions are also allowed to read data for which the executing site is not the primary or if transaction are distributed (i.e., they update data of different primaries). In this case, the scenario is similar to that in update everywhere server architectures and it is not enough that primaries send changes in FIFO order but a total order is necessary.

Discussion. Lacking a voting phase, this type of protocols are naturally cold-standby since the primary has no way of waiting for the secondaries to apply the changes. The primary can implement a 2-safe approach by not committing the transaction until the communication system guarantees the transaction will be delivered at the backups. This ensures that the protocol is 2-safe. If the primary commits the transaction without waiting, then the protocol is 1-safe.

If the backups are passive, that is, they do not do anything but installing the changes sent by the primary, determinism simply requires to install the changes in the order in which they arrive from the primary. If the backups are active and are executing transactions on their own behalf then there must be some rules to prevent inconsistencies. These rules can be summarised as follows: the local serialisation order cannot contradict the order in which the remote transactions arrive. By ensuring this, all sites produce conflict equivalent histories.

4.2.2 Primary Copy – Constant Interaction – Voting

Description. The introduction of a voting phase allows us to ensure that both the primary and the backups install the updates. Since the 2-safe property can be achieved independently of the voting phase, there is no point in using the voting phase for atomicity purposes. Instead, it is used to enforce hot-standby behaviour (see Figure 8):

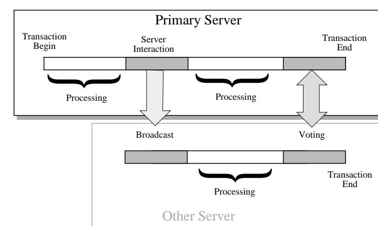

Figure 8. Primary copy, constant interaction, voting

① The transaction is executed at the primary.
② When the transaction terminates, the corresponding log records are broadcast to all backups.
③ The primary initiates a 2PC protocol.
④ The transaction is installed and committed at all sites.

Requirements. Compared to *primary copy – constant interaction – non-voting*, the requirements do not change by the introduction of the voting phase. However, since the primary waits until all backups have installed the transaction, the system is hot-standby.

Discussion. The nature of the broadcast depends on what has to be achieved. In principle, since the voting phase is any-

way done via 2PC, there is no requirement for the broadcast primitive used when the transaction is sent to all backups. If there is any problem, the transaction will abort during the 2PC. The 2PC can be greatly optimised if used only as a synchronisation point and not to guarantee atomic commitment. It is an open research question how to balance these two aspects in terms of cost and overhead.

If the backups are active, the use of 2PC allows to minimise the scheduling constraints. However, experience shows that minimising these constraints results in high abort rates. Thus, it is probably best to use total order broadcast and locally follow the delivery oder to avoid unnecessarily high abort rates.

4.2.3 Primary Copy – Linear Interaction – Non-voting

Description. Waiting until the transaction ends in order to propagate the changes causes long response times if the primary waits for the other sites (2-safe or hot-standby). The protocol could be faster if the backups work in parallel to the primary. In order to do this, the primary sends operations as they are executed, thereby allowing the backups to start doing some work. If no voting phase is involved, the protocol is as follows (see Figure 9):

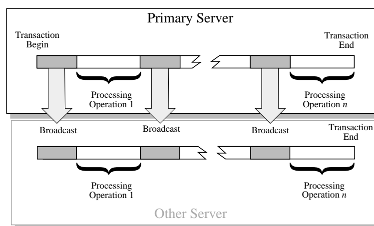

Figure 9. Primary copy, linear interaction, non-voting

① The transaction starts at the primary.
② Read operations are executed locally.
③ The results of write operations are broadcast to the backups.
④ A termination message indicates the end of the transaction.

Requirements. Since the backups receive operations and not transactions, one has to be more careful about the order in which changes are installed. In the case of passive backups, if the primary produces correct histories and sends operations in serialisation order, then FIFO delivery is enough to guarantee correctness. In general, since what is being sent to the backups are log records and log records are produced in serialisation order, the primary does not need to make any extra effort to ensure this property. If the backups are active and transactions may access data across primaries, determinism is again achieved by relying on total order. By serialising according to this total order, overall correctness is assured.

The termination message must be sent uniformly to all sites and the primary must wait until the message is received in order to ensure 2-safe behavior. Otherwise, the protocol is 1-safe. Since there is no voting, whether the protocol is hot- or cold-standby depends on whether the backups install the changes or they only save them to disk.

Discussion. Sending the operation as they are executed at the primary allows the backups to work in parallel but introduces a significant message overhead. Transactions typically have 20 update operations. Thus, to sustain a throughput of 100 transactions per second, the communication system must be capable of supporting a traffic of over 2000 broadcasts per second across the system. In practice, this is likely to be the biggest bottleneck when using this type of protocols.

4.2.4 Primary Copy – Linear Interaction – Voting

Description. As above, the purpose of introducing a voting phase is to ensure hot-standby behaviour (Figure 10):

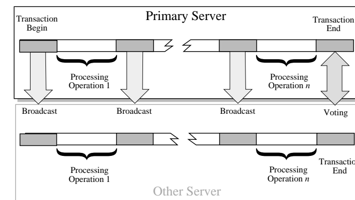

Figure 10. Primary copy, linear interaction, voting

① The transaction starts at the primary.
② Read operations are executed locally.
③ The results of write operations are broadcast to the backups.
④ The primary starts a 2PC protocol.
⑤ The transaction is installed and committed at all sites.

Requirements. Compared to non-voting, correctness is not affected by the voting phase. Nevertheless, active backups are free to abort any transaction since they can propagate this decision during the 2PC phase.

Discussion. As above, the use of 2PC at the end of each transaction removes any requirements for the broadcast primitive. In fact, this protocol is very similar to traditional replication protocols, and the discussions in Section 4.1 in the context of voting techniques also apply here.

5. Discussion

One of the goals of our classification effort was to identify the trade-off of the various replication techniques in order to identify the most promising approaches in terms of scalability and efficiency. When lazy techniques (those that do not ensure one-copy serialisability) and eager techniques (those that ensure one-copy serialisability) are usually compared [13], the comparison is with an expensive eager technique: update everywhere, linear interaction and a voting phase. This makes the comparison between lazy and eager techniques unfair. Indeed, a lazy technique will always perform better than an eager technique since it does not involve any message overhead during the execution of a transaction. However, eager replication can be much more efficient than currently implemented in commercial systems if adequate techniques are applied. We believe that eager replication, with its consistency guarantees and its flexibility, can be an attractive alternative to lazy replication. The classification provided shows how all the different alternatives compare and allows us to draw clear conclusions.

5.1. Server Architecture: Primary Copy vs. Update Everywhere

As already pointed out, update everywhere is a more elegant solution in that, in theory, it does not introduce bottlenecks. Thus, it may come as a surprise that most replication protocols used in practice are primary copy techniques. However, there are good reasons for this.

Update everywhere does not necessarily distribute the load among sites. Since the data is replicated, all sites need to perform the updates anyway. This means that unless there is a significant amount of read operations in the overall load (read operations being local), the system might not scale up as more server nodes are added. One way to improve the performances of update everywhere is to preprocess operations at one site and send the results to the other sites. That way, the processing does not need to be done everywhere. Once such mechanisms are in place, update everywhere becomes a more attractive solution since it is more robust to failures and facilitates distributing the load among the sites.

5.2. Server Interaction: Constant vs. Linear

The number of messages exchanged per transaction is a key aspect of any replication protocol. As pointed out before, sending one message per operation can quickly lead to unacceptable traffic rates. In addition, these messages need to be processed at each site, which significantly increases the load. Finally, since operations arrive at different points in time, coordinating their execution so that the overall result is correct is much more complicated.

It is a good rule of thumb to say that the less messages exchanged per transaction, the better. For instance, protocols based on linear interaction in combination with update everywhere are largely infeasible in databases. It is exactly this type of protocols that have been heavily criticised in the database community as unrealistic [13]. In the primary copy case, things are a bit different but the consequences of how many messages are exchanged are not negligible. In particular, sending all updates in one message at the end of the transaction can help to propagate the changes of those transactions that actually commit (sending updates on a per operation basis implies that operations that later will be aborted nevertheless contribute to the overall overhead).

Exchanging one message per transaction, however, introduces its own problems. These protocols work especially well for service requests where the data to be accessed is known in advance. In this case implementation is straightforward and abort rates are small. However, for ordinary transactions, some form of optimism must be used to first execute the transaction at the delegate server and then determine the serialisation order. If the conflict rates are high, this optimism might result in high abort rates.

5.3. Transaction Termination: Voting vs. Non-voting Techniques

Non-voting techniques are more demanding in terms of determinism requirements than voting techniques. With non-voting protocols, each server must independently guarantee the same serialisation as that of other servers. The typical way to do this is to use the total order as a guideline. In general, if two transactions conflict, their serialisation order will be that indicated by the total order. Depending on the protocol, sites need to known different things in order to ensure global correctness without voting. There are protocols where the whole transaction (read operations included) is sent. In these protocols, each site performs the equivalent of global scheduling for the whole system and, as long as this scheduling is deterministic, correctness is guaranteed. This determinism can be implemented by following the total order to serialise transactions.

In terms of voting techniques we have considered two possibilities, one of them is based on 2PC and another based on a confirmation message sent by the delegate or primary copy to indicate whether the transaction can be committed or must be aborted. The confirmation message is needed when only the delegate server (or the primary copy) of a transaction can unilaterally decide on the outcome of the transaction. However, remote sites must still behave deterministically in such a way that they must be able to obey the commit/abort decision of the delegate server.

When 2PC is used, each server can reject any transaction thus relaxing the determinism requirements since there is always a chance to resolve things during the 2PC. Unfortunately, it has been shown that in these cases, the coordination overhead is much higher, and according to [13], conflict, abort and deadlock rates can quickly become a bottleneck. Additionally, when voting is also used to provide atomicity, it can only take place when all sites have completely executed the transaction. This means, that the delegate server waits for the slowest of all replicas to finish processing before returning the result to the client, increasing transaction response times considerably.

5.4. Conclusion

Comparing the characteristics of these families of protocols several conclusions can be drawn. First, update everywhere has a good potential of distributing the load among the sites. Second, since linear interactions seem to be a significant source of overhead, realistically speaking, the only options left are approaches based on constant interactions. This is an important conclusion from our classification efforts. Database designers have not considered protocols on the UE-CI-V [1] or UE-CI-NV [2] categories. These seem to be the most promising approaches to eager replication (or primary copy with active backups, which has similar behavior and demand as update everywhere). Recent results seem to support this claim [28, 29, 19, 18], which is also strengthen by current developments that show how to reduce some of the overhead of group communication [30, 20, 27]. Regarding determinism, a judicious choice of the determinism point helps in designing protocols but, with minimal additional cost, there is always the possibility of using voting strategies where the determinism requirements are greatly reduced.

As part of future work, we are currently developing tools that will allow a quantitative evaluation of the different categories analysed. This will helps us to better understand the cost of the different approaches. Parallel to this, we will continue

[1] Update Everywhere – Constant Interaction – Voting.

[2] Update Everywhere – Constant Interaction – Non Voting.

exploring the space of solutions included in the UE-CI-V and UE-CI-NV categories.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), 1997.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] P.A. Bernstein, D.W. Shipman, and J.B. Rothnie. Concurrency control in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems*, 5(1):18–51, March 1980.

[4] A. Bhide, A. Goyal, H. Hsiao, and A. Jhingran. An efficient scheme for providing high availability. In *Proceedings of 1992 SIGMOD International Conference on Management of Data*, pages 236–245, May 1992.

[5] A. Borr. Robustness to crash in a distributed database: A non shared-memory multi-processor approach. In *Proceedings of* 10th *VLDB Conference*, Singapore, 1984.

[6] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–239, June 1992.

[7] S. Ceri, M. Houtsma, A. Keller, and P. Samarati. A classification of update methods for replicated databases. Technical Report CS-TR-91-1392, Stanford University, Computer Science Departement, May 1994.

[8] S. W. Chen and C. Pu. A structural classification of integrated replica control mechanisms. Technical Report CUCS-006-92, Columbia University, Departement of Computer Science, New York, NY 10027, 1992.

[9] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[10] S.B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[11] A. W. Fu and D. W. Cheung. A transaction replication scheme for a replicated database with node autonomy. In *Proceedings of the International Conference on Very Large Databases*, Santiago, Chile, 1994.

[12] H. Garcia-Molina and C. A. Polyzois. Two epoch algorithms for disaster recovery. In *Proceedings of* 16th *VLDB Conference*, pages 222–230, Brisbane, Australia, 1990.

[13] J. N. Gray, P. Helland, and and D. Shasha P. O'Neil. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996. SIGMOD.

[14] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[15] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[16] IBM, New Orchard Road, Armonk, NY 10504 (USA). *DB2: Replication Guide and Reference*, June 1999. Number SC26-9642-00.

[17] Information & Communcations Systems Research Group, ETH Zürich and Laboratoire de Systèmes d'Exploitation (LSE), EPF Lausanne. *DRAGON: Database Replication Based on Group Communication*, May 1998. http://www.inf.ethz.ch/department /IS/iks/research/dragon.html.

[18] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*. (to appear).

[19] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the* 18th *International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 1998.

[20] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the International Conference on Distributed Computing Systems*, Austin, Texas, June 1999.

[21] A. Kumar and A. Segev. Cost and availability tradeoffs in replicated concurrency control. *ACM Transactions on Database Systems*, 18(1):102–131, March 1993.

[22] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Potzulo, and B. W. Wade. Notes on distributed databases. Technical Report RJ2571(33471), IBM, San Jose Research Laboratory, 1979.

[23] M.C. Little and S.K. Shrivastava. Understanding the role of atomic transactions and group communications in implementing peristent objects. In *Eighth International Workshop on Persistent Object Systems: Design Implementation and Use*, August 1998.

[24] J. Lyon. Design considerations in replicated database systems for disaster protection. In *Proceedings of IEEE Compcon*, 1988.

[25] J. Lyon. Tandem's remote data facility. In *Proceedings of IEEE Compcon*, 1990.

[26] Oracle Corporation, 500, Oracle Parkway, Redwoord City, CA 94065. *Oracle*$8i^{tm}$ *Advanced Replication*, November 1998. Oracle Technical White Paper.

[27] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[28] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the* 16th *Symposium on Reliable Distributed Systems (SRDS-16)*, Durham, North Carolina, USA, October 1997.

[29] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.

[30] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the* 12th *International Symposium on Distributed Computing (DISC'98, formerly WDAG)*, September 1998.

[31] F. Pittelli and H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems*, 7(1):25–60, February 1989.

[32] S. Rangaranjan, S. Setia, and S.K. Tripathi. A fault-tolerant algorithm for replicated data management. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1271–1282, December 1995.

[33] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[34] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18th *IEEE International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, Amsterdam, The Netherlands, May 1998.

[35] D. B. Terry, K. Petersen, M. J. Spreizer, and M. M. Theimer. The case for non-transparent replication: Example from Bayou. *Bulletin of the Technical Commitee on Data Engineering*, 4(21):12–20, December 1998.

[36] O. Theel and H. Pagnia. Optimal replica control protocols exhibit symmetric operation availabilities. In *Proc. of the Int. Symp. on Fault-Tolerant Computing FTCS*, 1998.

[37] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[38] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of* 20th *International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., April 2000. IEEE Computer Society Los Alamitos California.

[39] U. G. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the* 14th *IEEE Symposium on Reliable Distributed Systems (SRDS-14)*, Bad Neuenahr, Germany, September 1995.