Understanding Replication in Databases and Distributed Systems

M. Wiesmann, F. Pedone, A. Schiper
Operating Systems Laboratory
Swiss Federal Institute of Technology (EPFL)
IN-Ecublens, CH-1015 Lausanne
{wiesmann,pedone,schiper}@lsemail.epfl.ch
http://lsewww.epfl.ch/

B. Kemme, G. Alonso
Institute of Information Systems
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, CH-8092 Zürich
{kemme,alonso}@inf.ethz.ch
http://www.inf.ethz.ch/department/IS/iks/

Abstract

Replication is an area of interest to both distributed systems and databases. The solutions developed from these two perspectives are conceptually similar but differ in many aspects: model, assumptions, mechanisms, guarantees provided, and implementation. In this paper, we provide an abstract and "neutral" framework to compare replication techniques from both communities in spite of the many subtle differences. The framework has been designed to emphasize the role played by different mechanisms and to facilitate comparisons. With this, it is possible to get a functional comparison of many ideas that is valuable for both didactic and practical purposes. The paper describes the replication techniques used in both communities, compares them, and points out ways in which they can be integrated to arrive to better, more robust replication protocols.

Keywords Replication protocols, database replication, replication comparison, fault tolerant systems, group communication, distributed systems

Technical Areas Distributed Fault-Tolerant Systems and Distributed Algorithms

Contact Author Fernando Pedone (pedone@lsemail.epfl.ch)

1 Introduction

Replication has been studied in many areas, especially in distributed systems (mainly for fault tolerance purposes) and in databases (mainly for performance reasons). In these two fields, the techniques and mechanisms used are similar, and yet, comparing the protocols developed in the two communities is a frustrating exercise that many researchers have unsuccessfully attempted. Due to the many subtleties involved, mechanisms that are conceptually identical, end up being very different in practice. As a result, it is very difficult to take results from one area and apply them in the other. In the last few years, as part of the DRAGON project [Dra98], we have devoted our efforts to enhance 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 [AAAS97, PGS97, PGS98] and used as part of the transaction manager to guarantee serialisable execution of transactions over replicated data [KA98, KA99]. We have also shown how some of the overheads associated with group communication can be hidden behind the cost of executing transactions, thereby greatly enhancing performance and removing one of the serious limitations of group communication primitives [KPAS99a]. This work has proven the importance of and the need for a common understanding of the replication protocols used by the two communities.

In this paper, we present a model that allows to compare and distinguish existing replication protocols in databases and distributed systems. We start by introducing a very abstract replication protocol representing what we consider to be the key phases of any replication strategy. Using this abstract protocol as the base line, we analyse a variety of replication protocols from both databases and distributed systems, and show their similarities and differences. With these ideas, we parameterise the protocols and provide an accurate view of the problems addressed by each one of them. Providing such a classification permits to systematically explore the solution space and give a good baseline for the development of new protocols. While such work is conceptual in nature, we believe it is a valuable contribution since it provides a much needed perspective on replication protocols. However, the contribution is not only a didactic one but also eminently practical. In recent years, and in addition to our work, many researchers have started to explore the combination of database and distributed system solutions [RTKA96, SAA98, PMS99, HAA99]. The results of this paper will help to show which protocols complement each other and how they can be combined.

The paper is organised as follows. Section 2 introduces our replication functional model and discusses some basis for our comparison. Section 3 and Section 4 present replication protocols in distributed systems and databases, respectively. In both cases, instead of concentrating on specific protocols, we focus on more general classes of replication algorithms. Section 5 refines the discussion presented in Section 4 for more complex transaction models. Section 6 discusses the different aspects of the paper and gives an conclusion.

2 Replication as an Abstract Problem

Replication in databases and distributed systems rely on different assumptions and offer different guarantees to the clients. Therefore, to understand its forms, replication must be seen as an abstract problem. In this section, we discuss the context of replication in databases and distributed systems, and introduce a functional model of replication.

2.1 Replication Context

Hereafter, we assume that the system is composed of a set of *replicas* over which operations must be performed. The operations are issued by *clients*. Communication between different system components (clients and replicas) takes place by exchanging messages.

In this context, distributed systems distinguish between the *synchronous* and the *asynchronous* system model. In the synchronous model there is a known bound on the relative process speed and on the message transmission delay, while no such bounds exist in the asynchronous model. The key difference is that the synchronous system allows *correct crash detection*, while the asynchronous system does not (i.e., in an asynchronous system, when some process p thinks that some other process q has crashed, q might in fact not have crashed). Incorrect crash detection makes the development of replication algorithm more difficult. Fortunately, much of the complexity can be hidden in the so called *group communication primitives*. This is the approach we have taken in the paper (see Section 3.1).

Databases are not concerned by the fondamental differences between synchronous and asynchronous systems for the following reason: databases accept to live with *blocking* protocols (a protocol is said to be blocking if the crash of some process may prevent the protocol from terminating). Blocking protocol are even simpler to design than non blocking protocols based on the synchronous model. Distributed systems usually look for non-blocking protocols.

This reflects another fundamental difference between distributed systems and database replication protocols. It has been shown that the specification of every problem can be decomposed into *safety* and *liveness* properties [AS87][1]. Database protocols do not treat liveness issues formally, as part of the protocol specification. Indeed, the properties ensured by transactions (Atomicity, Consistency, Isolation, Durability) [GR93] are all *safety* properties. However, because databases accept to live with blocking protocols, liveness is not an issue. For the purpose of this paper, we concentrate on safety properties.

Finally, database replication protocols may admit, in some cases, operator intervention to solve abnormal cases, like the failure of a server and the appointment of another one (a way to circumvent blocking). This is usually not done in distributed system protocols, where the replacement of a replica by another is integrated into the protocol (non-blocking protocols).

2.2 Functional Model

A replication protocol can be described using five generic phases. These phases represent important steps in the protocol and will be used to characterise the different approaches. As we will later show, some replication techniques may skip some phases, order them in a different manner, iterate over some of them, or merge them into a simpler sequence. Thus, the protocols can be compared by the way they implement each one of the phases

[1]A safety property says that nothing bad ever happens, while a liveness property says that something good eventually happens.

and how they combine the different phases. In this regard, an abstract replication protocol can be described as a sequence of the following five phases (see Figure 1).

1. **Request (RE):** the client submits an operation to one (or more) replicas.
2. **Server coordination (SC):** the replica servers coordinate with each other to synchronise the execution of the operation.
3. **Execution (EX):** the operation is executed on the replica servers.
4. **Agreement coordination (AC):** the replica servers agree on the result of the execution.
5. **Response (END):** the outcome of the operation is transmitted back to the client.

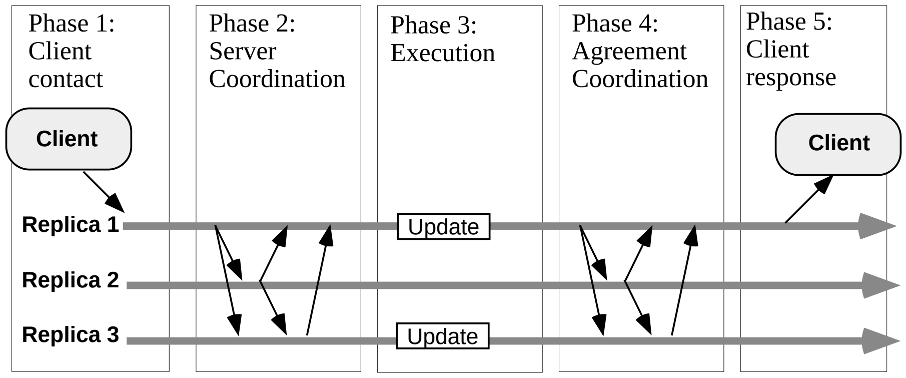

Figure 1: Functional model with the five phases

This functional model represents the basic steps of replication: submission of an operation, coordination among the replicas (e.g., to order concurrent operations), execution of the operation, further coordination among the replicas (e.g., to guarantee atomicity), and response to the client. The differences between protocols arise due to the different approaches used in each phase which, in some cases, obviate the need for some other phase (e.g., when messages are ordered based on an atomic broadcast primitive, the agreement coordination phase is not necessary since it is already performed as part of the process or ordering the messages).

Within this framework, we will first consider transactions composed of a single operation. This can be a single read or write operation, a more complex operation with multiple parameters, or an invocation on a method. A more advanced transaction model will be considered in Section 5. Although restrictive at first glance, this model is adopted by some database vendors, to handle web documents and stored procedures.

Request Phase. During the request phase, a client submits an operation to the system. This can be done in two ways: the client can directly send the operation to all replicas or the client can send the operation to one replica which will them send the operation to all others as part of the server coordination phase.

This distinction, although apparently simple, already introduces some significant differences between databases and distributed systems. In databases, clients never contact all replicas, and always send the operation to one copy. The reason is very simple: replication should be transparent to the client. Being able to send an operation to all replicas will imply the client has knowledge about the data location, schema, and distribution which is not practical for any database of average size. This is knowledge intrinsically tied to the database nodes, thus, client must always submit the operation to one node which will then send it to all others. In distributed systems, however, a clear distinction is made between replication techniques depending on whether the client sends the operation directly to all copies (e.g. active replication) or to one copy (e.g. passive replication).

It could be argued that in both cases, the request mechanisms can be seen as contacting a *proxy* (a database node in one case, or a communication module in the other), in which case there are no significant differences between the two approaches. Conceptually this is true. Practically, it is not a very helpful abstraction because of its implications as it will be discussed below when the different protocols are compared. For the moment being, note that distributed systems deal with *processes* while database deal with *relational schemas*. A list of processes is simpler to handle that a database schema, i.e., a communication module can be expected to be able to handle a list of processes but it is not realistic to assume it can handle a database schema. In particular, database replication requires to understand the operation that is going to be performed while in distributed systems, operation semantics usually play no role.

Finally, distributed systems distinguish between deterministic and non-deterministic replica behaviour. Deterministic replica behaviour assumes that when presented with the same operations in the same order, replicas will produce the same results. Such an assumption is very difficult to make in a database. Thus, if the different replicas have to communicate anyway in order to agree on a result, they can as well exchange the actual operation. By shifting the burden of broadcast the request to the server, the logic necessary at the client side is greatly simplified at the price of (theoretically) reducing fault tolerance. If fault tolerance is necessary, a back up system can be used, but this is totally transparent to the client.

Server Coordination Phase. During the server coordination phase, the different replicas try to find an order in which the operations need to be performed. This is the point where protocols differ the most in terms of ordering strategies, ordering mechanisms, and correctness criteria.

In terms of ordering strategies, databases order operations according to data dependencies. That is, all operations must have the same data dependencies at all replicas. It is because of this reason that operation semantics play an important role in database replication: an operation that only reads a data item is not the same as an operation that modifies that data item since the data dependencies introduced are not the same in the two cases. If there are no direct or indirect dependencies between two operations, they do not need to be ordered because the order does not matter. Distributed systems, on the other hand, are commonly based on very strict notions of ordering. From causality, which is based on *potential* dependencies without looking at the operation semantics, to total order (either causal or not) in which all operations are ordered regardless of what they are.

In terms of correctness, database protocols use serializability adapted to replicated scenarios: one-copy serializability [BHG87]. It is possible to use other correctness criteria[KA98] but, in all cases, the basis for correctness are data dependencies. Distributed systems use *linearisability* and *sequential consistency* [AW94]. Linearisability is strictly stronger than sequential consistency. Linearisability is based on *real-time* dependencies, while sequential consistency only considers the order in which operations are performed on every individual process. Sequential consistency allows, under some conditions, to read *old values*. In this respect, sequential consistency has similarities with one-copy serializability, but strictly speaking, the two consistency criteria are different. The distributed system replication techniques presented in this paper all ensure linearisability.

Execution Phase. The execution phase represents the actual performing of the operation. It does not introduce many differences between protocols, but it is a good indicator of how each approach treats and distributes the operations. This phase only represents the actual execution of the operation, the applying of the update is typically done in the Agreement Coordination Phase, even though applying the update to other copies may be done by re-executing the operations.

Agreement Coordination Phase. During this phase, the different replicas make sure that they all do the same thing. This phase is interesting because it brings up some of the fundamental differences between protocols. In databases, this phase usually corresponds to a Two Phase Commit Protocol (2PC) during which it is decided whether the operation will be committed or aborted. This phase is necessary because in databases, the Server Coordination phase takes care only of ordering operations. Once the ordering has been agreed upon, the replicas need to ensure everybody agrees to actually committing the operation. Note that being able to order the operations does not necessarily mean the operation will succeed. In a database, there can be many reasons why an operation succeeds at one site and not at another (load, consistency constraints, interactions with local operations). This is a fundamental difference with distributed systems where once an operation has been successfully ordered (in the Server Coordinator phase) it will be delivered (i.e., "performed") and there is no need to do any further checking.

Client Response Phase. The client response phase represents the moment in time when the client receives a response from the system. There are two possibilities: either the response is sent only after everything has been settled and the operation has been executed, or the response is sent right away and the propagation of changes and coordination among all replicas is done afterwards. In the case of databases, this distinction leads to $1)$ the so called eager or synchronous (no response until everything has been done) and $2)$ lazy or asynchronous (immediate response, propagation of changes is done afterwards) protocols. In the distributed systems case, the response takes place only after the protocol has been executed and no discrepancies may arise.

The client response phase is of increasing importance given the proliferation of applications for *mobile* users, where a copy is not always connected to the rest of the system and it does not make sense to wait until updates take place to let the user see the changes made.

3 Distributed Systems Replication

In this section, we describe the model and the communications abstractions used by replication protocols in distributed systems, and present four replication techniques that have been proposed in the literature in the context of distributed systems.

3.1 Replication Model and Abstractions

We consider a distributed system modelled as a set of services implemented by servers processes and invoked by clients processes. The specification of the service defines the set of invocations that can be issued by the clients. Each server process has a local state that is modified through invocations. We consider that invocations modify the state of a server in an atomic way, that is, the state changes resulting from an invocation are not applied partially. The isolation between concurrent invocations is the responsibility of the server, and is typically achieved using some local synchronisation mechanism. This model is similar to "one operation" transactions in databases (e.g., stored procedure). In order to tolerate faults, services are implemented by multiple server processes or replicas.

To cope with the complexity of replication, the notion of *group* (of servers) and *group communication primitives* have been introduced [Bir93]. The notion of *group* acts as a logical addressing mechanism, allowing the client to ignore the degree of replication and the identity of the individual server processes of a replicated service. *Group communication primitives* provide one-to-many communication with various powerful semantics. These semantics hide much of the complexity of maintaining the consistency of replicated servers. The two main group communication primitives are *Atomic Broadcast* (or ABCAST) and *View Synchronous Broadcast* (or VSCAST). We give here an informal definition of these primitives. A more formal definition of ABCAST can be found in [HT93] and of VSCAST can be found in [SS93] (see also [BJ87, BSS91]).

Group communication properties can also feature FIFO order guarantees, that is, if a process broadcasts a message m before a message m', then no process delivers m' before m.

Atomic Broadcast (ABCAST). Atomic Broadcast provides *atomicity* and *total order*. Let m and m' be two messages that are ABCAST to the same group g of servers. The atomicity property ensures that if one member of g delivers m (respt. m'), then all (not crashed) members of g eventually deliver m (respt. m'). The order property ensures that if two members of g deliver both m and m', they deliver them in the same order.

View Synchronous Broadcast (VSCAST). The definition of View Synchronous Broadcast is more complex. It is defined in the context of a group g, and is based on the notion of *a sequence of views* $v_0(g), v_1(g), \ldots, v_i(g), \ldots$ of group g. Each view $v_i(g)$ defines the composition of the group at same time t, i.e. the members of the group that are perceived as being correct at time t. Whenever a process p in some view $v_i(g)$ is suspected to have crashed, or some process q wants to join, a new view $v_{i+1}(g)$ is installed, which reflects the membership change.

Roughly speaking, VSCAST of message m by some member of the group g currently in view $v_i(g)$ ensures the following property: if one process p in $v_i(g)$ delivers m before installing view $v_{i+1}(g)$, than no process installs view $v_{i+1}(g)$ before having first delivered m.

3.2 Active Replication

Active replication, also called the state machine approach [Sch90], is a non-centralised replication technique. Its key concept is that all replicas receive and process the same sequence of client requests. Consistency is guaranteed by assuming that, when provided with the same input in the same order, replicas will produce the same output. This assumption implies that servers process requests in a *deterministic* way.

Clients do not contact one particular server, but address servers as a group. In order for servers to receive the same input in the same order, client requests can be propagated to servers using an Atomic Broadcast. Weaker communication primitives can also be used if semantic information about the operation is known (e.g., two requests that commute do not have to be delivered at all servers in the same order).

The main advantage of active replication is its simplicity (e.g., same code everywhere) and failure transparency. Failures are fully hidden from the clients, since if a replica fails, the requests are still processed by the other replicas.

The determinism constraint is the major drawback of this approach. Although one might also argue that having all the processing done on all replicas consumes too much resources. Notice however, that the alternative, that is, processing a request at only one replica and transmitting the state changes to the others (see next section), in some cases may be much more complex and expensive than simply executing the invocation on all sites.

Figure 2 depicts the active replication technique using an Atomic Broadcast as communication primitive. In active replication, phases **RE** and **SC** are merged and phase **AC** is not used.

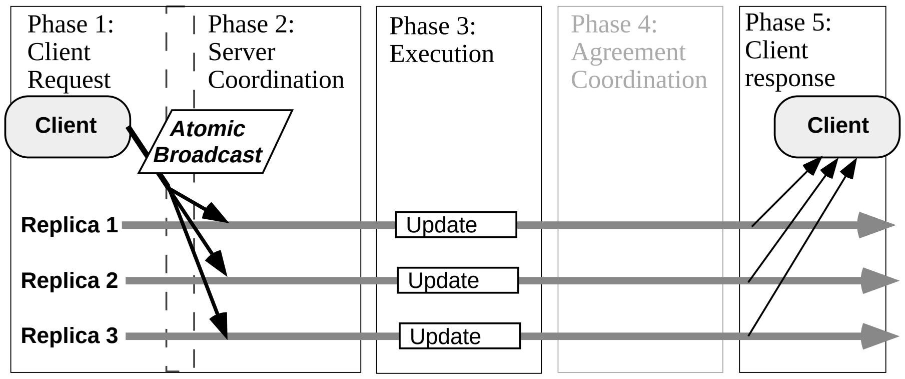

Figure 2: Active replication

The following steps are involved in the processing of an update request in the Active Replication, according to our functional model.

1. The client sends the request to the servers using an Atomic Broadcast.
2. Server coordination is given by the total order property of the Atomic Broadcast.
3. All replicas execute the request in the order they are delivered.
4. No coordination is necessary, as all replica process the same request in the same order. Because replica are deterministic, they all produce the same results.
5. All replica send back their result to the client, and the client typically only waits for the first answer (the others are ignored).

3.3 Passive Replication

The basic principle of passive replication, also called *Primary Backup* replication, is that clients send their requests to a primary, which executes the requests and sends update messages to the backups (see Figure 3). The backups do not execute the invocation, but apply the changes produced by the invocation execution at the primary (i.e., updates). By doing this, no determinism constraint is necessary on the execution of invocations, the main disadvantage of active replication.

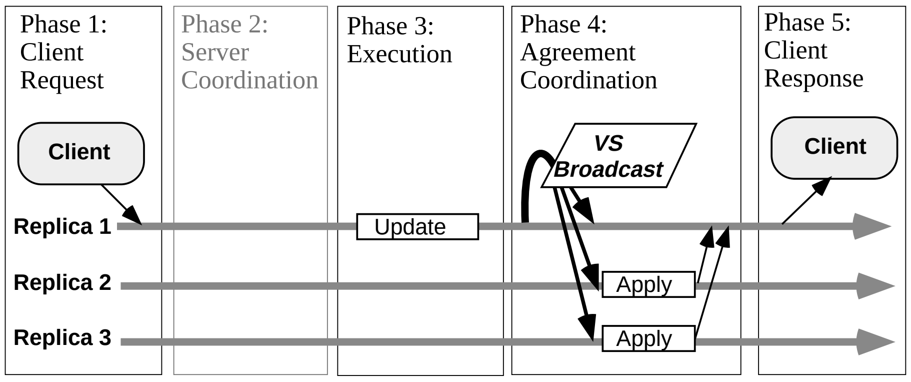

Figure 3: Passive replication

Communication between the primary and the backups has to guarantee that updates are processed in the same order, which is the case if primary backup communication is based on FIFO channels. However, only FIFO channels is not enough to ensure correct execution in case of failure of the primary. For example, consider that the primary fails before all backups receive the updates for a certain request, and another replica takes over as a new primary. Some mechanism has to ensure that updates sent by the new primary will be “properly” ordered with regard to the updates sent by the faulty primary. VSCAST is a mechanism that guarantees these constraints, and can usually be used to implement the primary backup replication technique [GS97].

Passive replication can tolerate non-deterministic servers (e.g., multi-threaded servers) and uses little processing power when compared to other replication techniques. However, passive replication suffers from a high reconfiguration cost when the primary fails.

The five steps of our framework are the following:

1. The client sends the request to the primary.
2. There is no initial coordination.
3. The primary executes the request.
4. The primary coordinates with the other replicas by sending the update information to the backups.
5. The primary sends the answer to the client.

3.4 Semi-Active Replication

Semi-active replication is an intermediate solution between active and passive replication. Semi-active replication does not require that replicas process service invocation in a deterministic manner. The protocol was originally proposed in a synchronous model [PCD91]. We present it here in a more general system model.

The main difference between semi-active replication and active replication is that each time replicas have to make a non-deterministic decision, a process, called the *leader*, makes the choice and sends it to the *followers*. Figure 4 depicts Semi-active replication. Phases **EX** and **AC** are repeated for each non deterministic choice.

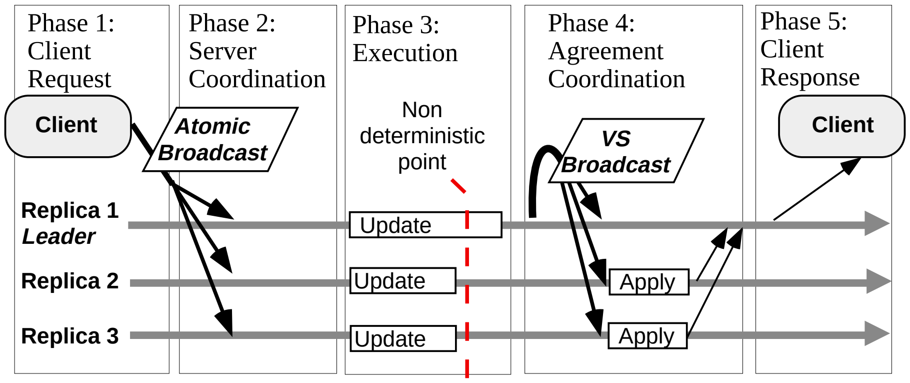

Figure 4: Semi-active replication

The following steps characterise semi-active replication, according to our framework.

1. The client sends the request to the servers using an Atomic Broadcast.
2. The servers coordinate using the order given by this Atomic Broadcast.

3. All replicas execute the request in the order they are delivered.
4. In case of a non deterministic choice, the *leader* informs the *followers* using the View Synchronous Broadcast.
5. The servers sends back the response to the client.

3.5 Semi-Passive Replication

Semi-passive replication [DSS98] is a variant of passive replication which can be implemented in the asynchronous model without requiring the view synchronous communication mechanism, i.e., without requiring the notion of views. The main advantage over passive replication is two allow for aggressive time-outs value to suspect crashed processes, without incurring a too important cost for incorrect failure suspicions. Because this technique has no equivalence in the context of database replication, we do not discuss it in detail. Roughly speaking, in semi-passive replication the Server Coordination (phase 2) and the Agreement Coordination (phase 4) are part of one single coordination protocol called *Consensus with Deferred Initial Values*.

3.6 Summary

Figure 5 summarises the different replication approaches in distributed systems, grouped according the following two dimensions: (1) failure transparency for clients, and (2) server determinism.

	Server Determinism Needed	Server Determinism **Not** Needed
Server Failure **Not** Transparent for the Client		**Passive**
Server Failure Transparent for the Client	**Active**	**semi-Active** **semi-Passive**

Figure 5: Replication in distributed systems

4 Database Replication

Replication in database systems is done mainly for performance reasons. The objective is to access data locally in order to improve response times and eliminate the overhead of having to communicate with other sites. If consistency needs to be guaranteed, this is only possible for read operations. Otherwise, both read and writes can be done locally, leaving consistency in the hands of the replication protocol. Fault tolerance is an issue but it is solved using back up mechanisms which, even being a form of replication, are entirely transparent to the clients.

4.1 Replication Model in Databases

A database can be seen as a collection of data items controlled by a database management system. A replicated database is thus a collection of databases that store copies of the same data item (for simplicity, we assume full replication). Hence, we distinguish a logical data item X and its physical copies X_i on the different sites. The basic unit of replication is the data item.

Clients access the data by submitting transactions to the database system. An operation, $o_i(X)$, of a transaction, T_i, can be either a read or a write access to a logical data item, X, in the database. This logical operation must then be translated to physical operations on the copies of the object. Moreover, a transaction is a unit of work that executes atomically, i.e., a transaction either commits or aborts its results on all participating sites. Furthermore, if transactions run concurrently they must be isolated from each other if they conflict. Two operations conflict if both access the same data item and one of them is a write. Isolation is provided by concurrency control mechanisms such as locking protocols [BHG87] which guarantee serializability. These protocols are extended to work in replicated scenarios and to provide 1-copy serializability, the accepted correctness criterion for database replication [BHG87].

From an architectural point of view, a client submits its transactions to only one database and, in general, it is connected only to this database. If a database server fails, active transactions (not yet committed or aborted) running on that server are aborted. Clients can then be connected to another database server and re-submit the transaction. The failure is seen by the client but, in return, the client's logic is much simpler. From a practical point of view, in any working system, failures are the exception so it makes sense to optimise for the situation when failures do not occur as databases do.

In this section, we will use a very simple form of transaction that consists of a single operation. This allows us to concentrate on the coordination and interaction steps and makes it possible to directly compare with distributed system approaches. The next section will refine this model to extended to normal transactions. Although the *single operation* approach may seem restrictive, it is actually used by many commercial systems in the form of *stored procedures*. A stored procedure resembles a procedure call and contains all the operations of one transaction. By invoking the stored procedure, the client invokes a transaction.

4.2 Replication Strategies

Gray et.al [GHPO96] have categorised database replication protocols using two parameters (see Figure 6). One is when update propagation takes place (eager vs. lazy) and the second is who can perform updates (primary vs. update-everywhere). In eager replication schemes, updates are propagated within the boundaries of a transaction, i.e., the user does not receive the commit notification until sufficient copies in the system have been updated. Lazy schemes, on the other hand, update a local copy, commit and only some time after the commit, the propagation of the changes takes place. The first approach provides consistency in a straightforward way but it is expensive in terms of message overhead and response time. Lazy replication allows a wide variety of optimisations, however, since copies are allowed to diverge, inconsistencies might occur.

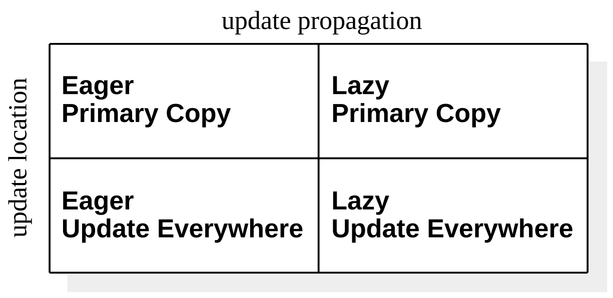

Figure 6: Replication in database systems

In regard to who is allowed to perform updates, the primary copy approach requires all updates to be performed first at one copy (the primary or master copy) and then at the other copies. This simplifies replica control at the price of introducing a single point of failure and a potential bottleneck. The update everywhere approach allows any copy to be updated, thereby speeding up access but at the price of making coordination more complex.

4.3 Eager Primary Copy Replication

In an eager primary copy approach, an update operation is first performed at a primary master copy and then propagated from this master copy to the secondary copies. When the primary has the confirmation that the secondary copies have performed the update, it commits and returns a notification to the user. Ordering of conflicting operations is determined by the primary site and must be obeyed by the secondary copies. Reading transactions can be performed on any site and reading transactions will always see the latest version of each object. Early solutions, e.g., distributed INGRES [AD76, Sto79], used this approach. Currently, it is only used for fault-tolerance in order to implement a hot-standby backup mechanism where a primary site executed all operations and a secondary site is ready to immediately take over in case the primary fails[2] [GR93, AKA$^+$96].

Figure 7 shows the steps of the protocol in terms of the functional model as it would be used in a hot stand-by back-up mechanism. The server coordination phase disappears since execution takes place only at the primary. The execution phase involves performing the transactions to generate the corresponding log records which are then sent to the secondary and applied. Then a 2PC protocol is executed during the agreement coordination phase. Once this finishes, a response is returned to the client.

[2]Note that the primary is still a single point of failure, such an approach assumes that a human operator can reconfigure the system so that the back-up is the new primary

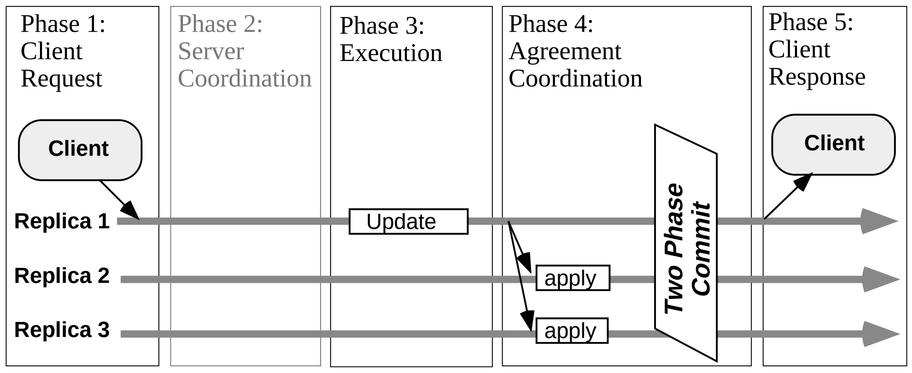

Figure 7: Eager primary copy

From here, it is easy to see that eager primary copy replication is functionally equivalent to passive replication with VSCAST. The only differences are internal to the Agreement Coordination phase (2PC in the case of databases and VSCAST in the case of distributed systems). This difference can be explained by the use of transactions in databases. As explained, VSCAST is used to guaranteed that operations are ordered correctly even after a failure occur. In a database environment, the use of 2PC guarantees that if the primary fails, all active transactions will be aborted. Therefore, there is no need to order operations from “before the failure” and “after the failure” since there is only one source and the different views cannot overlap with each other.

4.4 Eager Update Everywhere Replication

From a functional point of view there are two types of protocols to consider depending on whether they use distributed locking or atomic broadcast to order conflicting operations.

4.4.1 Distributed Locking Approach

When using distributed locking, a replica can only be accessed after it has been locked at all sites. For transactions with one operation, the replication control runs as follows (see Figure 8). The client sends the request to its local database server. This server sends a lock request to all other servers which grant or do not grant the lock. The lock request acts as the Server Coordination phase. If the lock is granted by all sites, we can proceed. If not, the transaction can be delayed and the request repeated some time afterwards. When all the locks are granted, the operation is executed at all sites. During the Agreement Coordination phase, a 2PC protocol is used to make sure that all sites commit the transaction. Afterwards, the client gets a response.

A comparison between Figures 4 and 8 shows that semi-active replication and eager update everywhere using distributed locking are conceptually similar. The differences arise from the mechanisms used during the Server Coordination and Agreement Coordination phases. In databases, Server Coordination takes place using 2 Phase

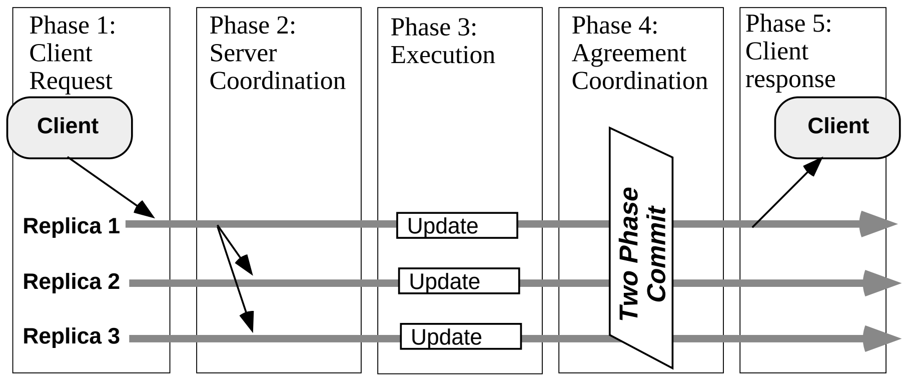

Figure 8: Eager update everywhere with distributed locking

Locking [BHG87] while in distributed systems this is achieved using ABCAST. The 2 Phase Commit mechanism used in the Agreement Coordination phase of the database replication protocol corresponds to the use of a VSCAST mechanism in the distributed systems protocol. Note that, if it could be assumed that databases are deterministic, the 2PC mechanism would not be needed and the protocol would be functionally identical to active replication (as shown below).

4.4.2 Data Replication based on Atomic Broadcast

The idea of using group communication primitives to implement database replication has been around for quite some time. However, it has not been until recently that the problem has been tackled with sufficient depth so as to provide realistic solutions [SR96, KA98, KPAS99a]. The basic idea behind this approach is to use the total order guaranteed by ABCAST to provide a hint to the transaction manager on how to order conflicting operations. Thus, the client submits its request to one database server which then broadcasts the request to all other database servers (note that in distributed systems, the client broadcasts the request directly to all servers). Instead of 2 Phase Locking, the server coordination is done based on the total order guaranteed by ABCAST and using some techniques to obtain the locks in a consistent manner at all sites [KA98, KPAS99b]. Then the operation is executed and a response sent back to the client. the following steps are involved in this approach (see Figure 9).

1. the client sends the request to the local server
2. the server forwards the request to all servers which coordinate using the order given by the Atomic broadcast.
3. the servers execute the transaction. If two operations conflict they are executed in the order of the atomic broadcast.
4. there is no coordination at this point.
5. the local servers sends back the response to the client.

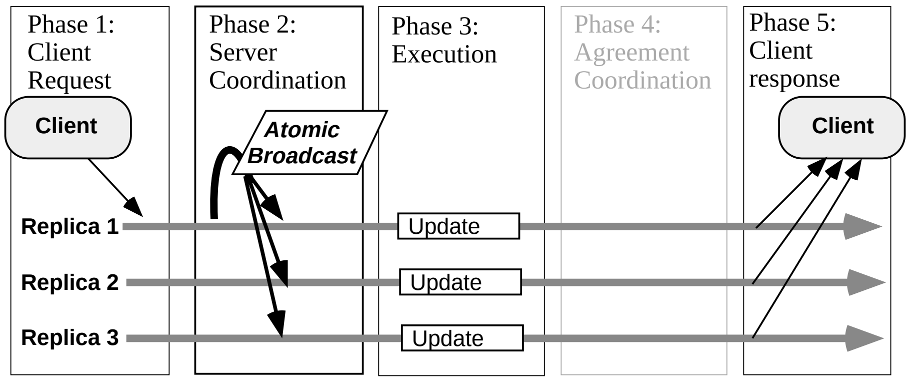

Figure 9: Eager update everywhere based on atomic broadcast

The similarities between active replication and eager update everywhere using ABCAST are obvious when Figures 2 and 9 are compared. The only significant difference is the interaction between the client and the system, as pointed out above. Regarding the determinism of the databases, a complete study of the requirements and the conditions under which ABCAST can be used for database replication and when an Agreement Coordination is necessary can be found in [KA98].

4.5 Lazy Primary Copy

Lazy replication avoids the synchronisation overhead of eager replication techniques by providing a response to the clients before there is any coordination between servers. Moreover, since the coordination will take place only afterwards, the client only needs to communicate with one server before the operation is executed and a response given. In the case of primary copy, all clients must contact the same server to perform an update. Thus, a lazy primary copy protocol can be seen as the sequence of phases shown in Figure 10.

The major aspect of lazy approaches is the Agreement Coordination phase. During this phase, the copies are brought to a consistent state by propagating all changes and deciding on how to apply them. In the case of primary copy, this phase is relatively straightforward in that any necessary coordination and ordering between transactions happens at the primary and the replicas need only to apply the changes as the primary propagates them.

4.6 Lazy Update Everywhere

Also the change to update everywhere approaches is not very big. Figure 11 shows the different steps. Some time after the transaction commits, the updates are propagated to the other sites. However, as in the case of eager update everywhere, coordination is much more complicated than with a primary copy approach. Since the other sites might have run conflicting transactions at the same time, the copies on the different site might not only be

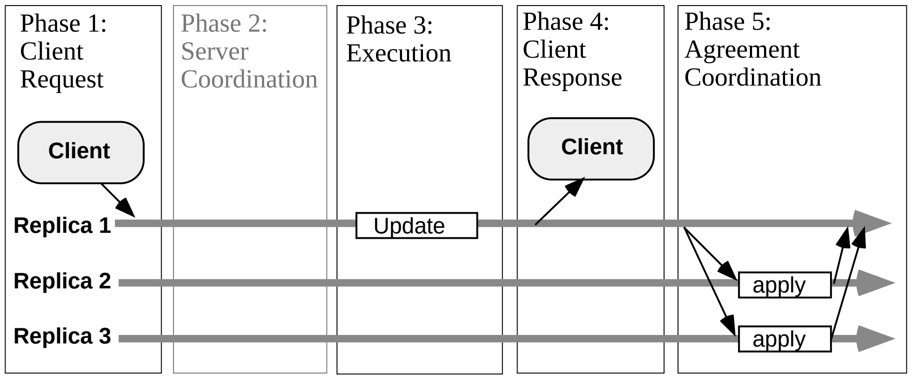

Figure 10: Lazy primary copy

stale but inconsistent. Reconciliation is needed to decide which updates are the winners and which transactions must be undone. There are some reconciliation schemes around, however, most of them are on a per object basis. This is enough in the case where one transaction consists of one operation, however, they are not sufficient when transactions consists of more operations on different objects.

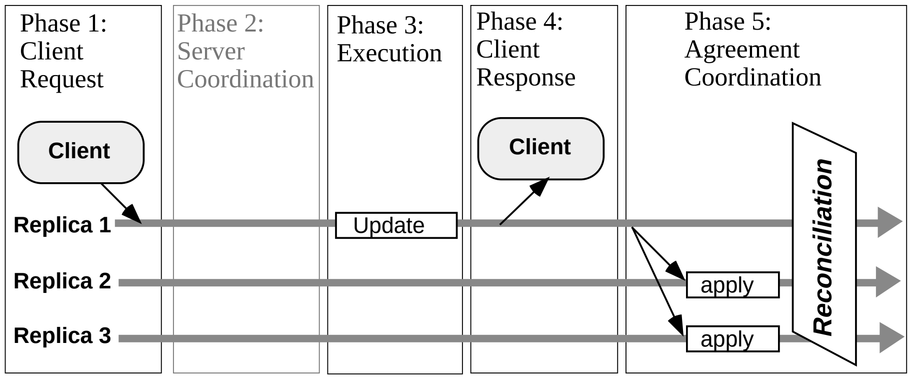

Figure 11: Lazy Update Everywhere

A straightforward solution in the case of our simple model is to run an Atomic Broadcast and determine the after-commit-order according to the order of the atomic broadcast.

Note that the concept of laziness, while existing in distributed systems approaches [RL92], is not widely used. This reflects the fact that those solutions are mainly developed for fault-tolerant purposes, making an eager approach obligatory. Lazy approaches, on the other hand, are a straightforward solution if performance is the main issue. Response times have to be short not allowing any communication within a transaction.

5 Transactions

In many databases, transactions are not one single operation or are not executed via a stored procedure. Instead, transactions are a partial order of read and write operations which are not necessarily available for processing at the same time. This has important consequences for replication protocols. In particular, the protocol has to deal more explicitly with the execution of the transaction, taking into account multiple operations. These new set of protocols have no equivalent in distributed systems since the notion of transaction is not that common in this community.

5.1 Transactions in the Functional Model

A transaction T_i is a partial order of read and write operations $o_i(X)$. These operations are executed over a logical data item and translated by the replication protocol into physical operations over the replicas. Operations conflict if they are from different transactions, access the same data item and at least one of them is a write. A *history* is a partial order of physical operations that includes all intra-transaction orderings and also orders all conflicting operations. A *serial history* is a history in which the execution of each transaction is not interleaved with the execution of other transactions. From here, a history is serialisable (correct) if it is conflict equivalent to a serial history. Conflict equivalence implies that the two histories are over the same set of operations and transactions and operations that conflict are ordered in the same way in both histories.

The fact that now a transaction has many operations and that those operations need to be properly ordered with respect to each other requires to modify the functional model. The modification involves introducing a loop including the Server Coordination and Execution phases or the Execution and Agreement Coordination phases, depending on the protocol used. The loop will be executed once for each operation that needs to be performed.

5.2 Eager Primary Copy Replication

In the case of primary copy, there is no need for server coordination. Hence, the loop will involve the Execution and the Agreement Coordination phase. In this loop an operation is performed at the primary copy and then the changes sent to the replicas. This is done for every operation and at the end, a new Agreement Coordination phase is executed in order to go through a 2PC protocol that will commit the transaction at all sites (Figure 12).

Note that the Agreement Coordination phases for each operation and that at the end use different mechanisms. If we compare this with the algorithm in Section 4.4.1, we notice that last phase is the same. For each operation except the last, it suffices to send the operation. In the final Agreement Coordination phase, a 2PC protocol is used to make sure all sites either commit or abort the transaction.

An alternative approach to this one is to use shadow copies and propagate the changes made by a transaction only after the transaction has completed (note that completed is not the same as committed!). If this approach is used, the resulting protocols is identical to that shown in Figure 7.

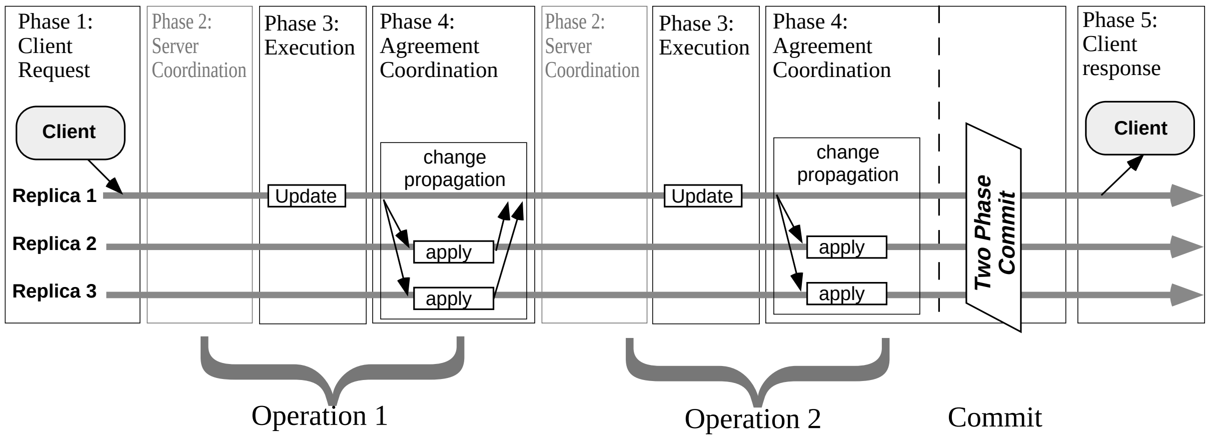

Figure 12: Eager primary copy approach for transactions

5.3 Lazy Primary Copy Replication

When using lazy replication, updates are not propagated until the transaction commits. Then all the updates performed by the transaction are sent as a unit. Thus, whether transactions have one or more operations does not make a difference for lazy replication protocols.

5.4 Eager update everywhere replication

We will again look at the two different approaches used to implement eager update everywhere replication.

5.4.1 Distributed Locking

In this case, a lock must be obtained for every operation in the transaction. This requires to repeat the Server Coordination and Execution phases for every operation. At the end, once all operations have been processed in this way, a 2PC protocol is used during the Agreement Coordination phase to commit or abort the transaction at all sites (Figure 13).

Note that the use of quorums is orthogonal to this discussion. Quorums only determine how many sites and which of them need to be contacted in order to obtain the locks. Independently of which sites participate, the phases of the protocol are the same. In an extreme case, read operations are local (read-one/write-all approach [BHG87]), which still requires the phases shown for all write operations.

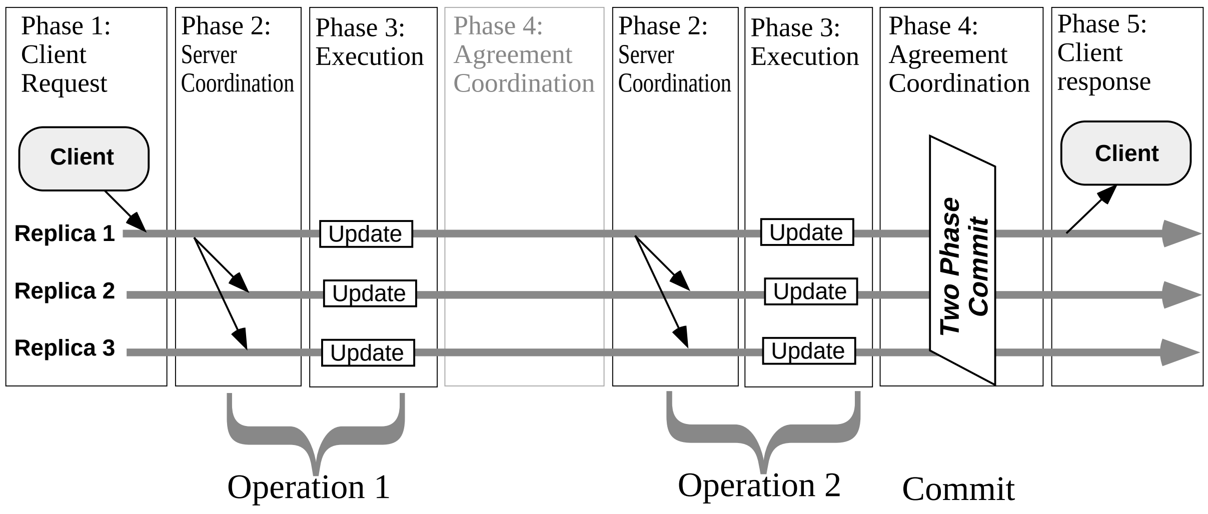

Figure 13: Eager update everywhere approach for transactions

5.4.2 Certification Based Database Replication

When using ABCAST to send the operations to all replicas, the resulting total order has no bearing on the serialisation order that needs to be produced. For this reason, it does not make much sense to use ABCAST to send every operation of a transaction separately. It makes sense, however, to use shadow copies at one site to perform the operations and then, once the transaction is completed, send all the changes in one single message [KA98]. Due to the fact that now a transaction manager has to unbundle these messages, the agreement coordination phase gets to be more complicated since it involves deciding whether the operations can be executed correctly. This can be seen as a *certification* step during which sites make sure they can execute transactions in the order specified by the total order established by ABCAST (Figure 14).

6 Discussion

This paper presents a general comparison of replication approaches used in the distributed system and database communities. Our approach was to first characterise replication algorithms using a generic framework. Our generic framework identifies five basics steps, and, although simple, allowed us to classify classical replication protocols described in the literature on distributed systems and databases.

Figure 15 summarises the combinations of the replication techniques presented in the paper that guarantee strong consistency (i.e., linearisability and one-copy serialisability). From Figure 15 we see that any replication technique that ensures strong consistency has either an **SC** and/or **AC** step before the **END** step.

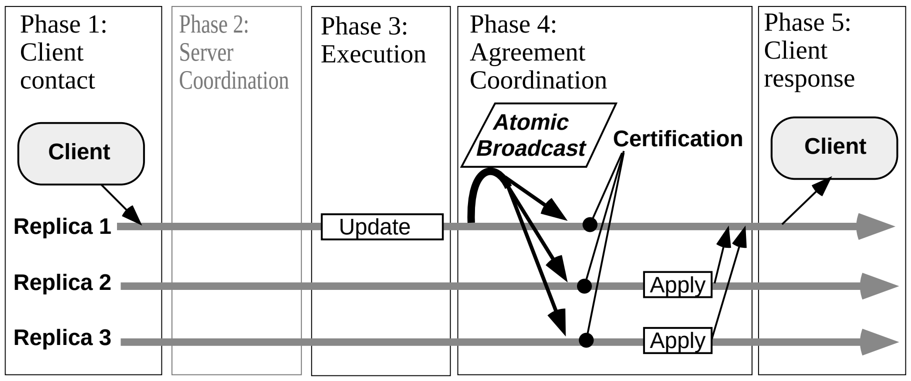

Figure 14: Certification based Database Replication

All techniques have at least one synchronisation step (**SC** or **AC**). If the execution step (**EX**) is deterministic, no synchronisation after **EX** is needed, as the execution will yield the same result on all servers. For the same reason, if only one server does the execution step, there is no need for synchronisation before the execution.

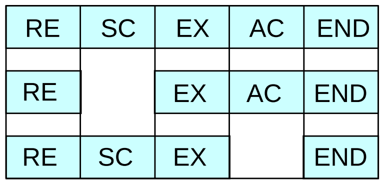

Figure 15: Possible combination of phases

Figure 16 summarises the different replication techniques in the case of single operation transactions.

Several conclusion can be drawn from this figure. First, primary copy and passive replication schemes share one common trait: they do not have an **SC** phase (since the primary does the processing, there is no need for early synchronisation between replicas). Furthermore, update everywhere replication schemes need the initial **SC** phase before an update can be executed by the replicas. The only exception are the Certification based techniques that use Atomic Broadcast (Sect. 5.4.2). Those techniques are *optimistic* in the sense that they do the processing without initial synchronisation, and abort transactions in order to maintain consistency. Finally, the difference between eager and lazy replication techniques is the ordering of the **AC** and **END** phases: in the eager technique, the **AC** phase comes first, while in the lazy technique, the **END** phase comes first.

Despite different models, constraints and terminologies, replication algorithms for distributed systems and databases bear several similarities. These similarities put into evidence the need for stronger cooperation between both communities. For example, replicated databases could benefit from the abstractions of distributed systems.

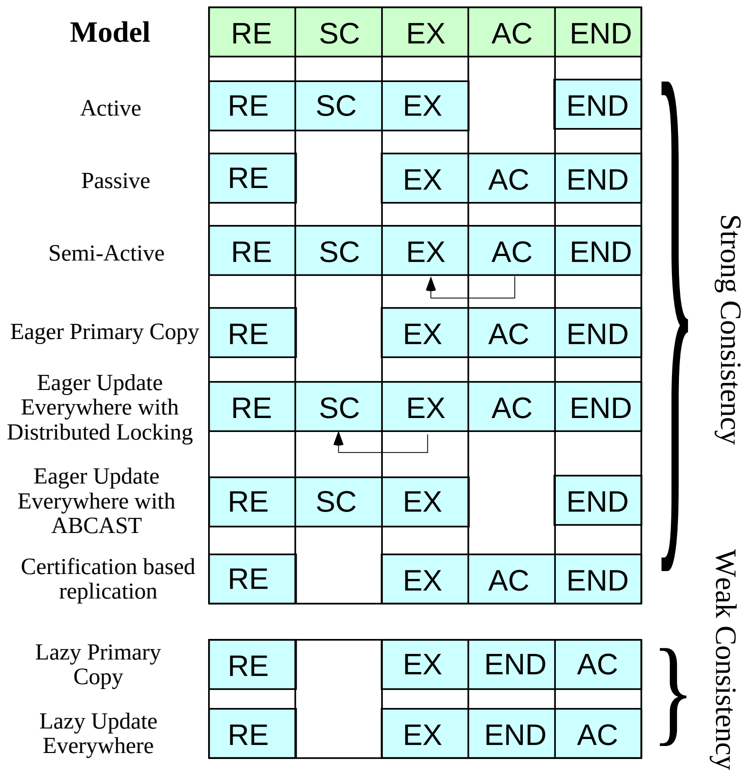

Figure 16: Synthetic view of approaches

Presently, we are planning a performance study of the different approaches, taking into account different workloads and failures assumptions.

References

[AAAS97] 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.

[AD76] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the International Conference on Software Engineering*, October 1976.

[AKA+96] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Advanced transaction models in the workflow contexts. In *Proceedings of the International Conference on Data Engineering*, New Orleans, February 1996.

[AS87] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[AW94] H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Bir93] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[BJ87] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on OS Principles*, pages 123–138, Austin, TX, USA, November 1987. ACM SIGOPS, ACM.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[Dra98] 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`.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.

[GHPO96] 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 173–82, Montreal, Canada, June 1996. SIGMOD. Microsoft Technical Report MSR-TR-96-17.

[GR93] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques.* Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *Proceedings of IEEE International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158–165, 1999.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. adwe, second edition, 1993.

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th Internationnal Conference on Distributed Computing Systems (ICDCS)*, Amsterdam, The Netherlands, May 1998.

[KA99] B. Kemme and G. Alonso. Transactions, messages and events: Merging group communication and database system. In *3rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island (Portugal), April 23–28, 1999. BROADCAST Esprit WG 22455.

[KPAS99a] 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, 1999. to appear.

[KPAS99b] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Using optimistic atomic broadcast in transaction processing systems. Technical report, Department of Computer Science, ETH Zürich, March 1999.

[PCD91] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in Delta-4*. *ACM Operating Systems Review, SIGOPS*, 25(2):122–125, April 1991.

[PGS97] 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.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.

[PMS99] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of the 25th International Conference on Very Large Databases*, Edinburgh - Scotland - UK, 7–10 September 1999.

[RL92] S. Ghemawat R. Ladin, B. Liskov. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[RTKA96] M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. Technical Report 983, Institut de Recherche en Informatique et Systèmes Aléatoires, February 1996.

[SAA98] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, Amsterdam, The Netherlands, May 1998.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[SS93] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS-13)*, pages 561–568, Pittsburgh, Pennsylvania, USA, May 1993. IEEE Computer Society Press.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188–194, May 1979.