CHAPTER 4

Replication Architecture

This chapter is concerned with the architecture of replicated database systems. The protocols described in the last chapter were kept at a rather high-level without giving any details of how they could actually be implemented in a real system. This chapter introduces the basic design alternatives that exist for putting a replication solution into practice. The task of architecting a system consists of deciding what functionality is provided, how this functionality is packaged into subsystems, and how these subsystems interface. Understanding the trade-offs between various engineering alternatives is important as they have an influence on non-functional attributes such as availability, scalability, maintainability, etc.

The most crucial design decision one has to make is to decide where to locate the replication logic. The replication module could be an integral part of the database engine or it could be located in a middleware component that resides between the clients and the unchanged database kernel. For middleware based systems there exists again various architectural alternatives, such as whether there is a centralized middleware component, or the middleware itself is decentralized. The choice of architecture has a fundamental impact on how replica control is correlated with concurrency control, how clients connect to the system, how the replication module interacts with the database, and how update processing is performed. All these issues are dealt with in the first section of this chapter.

A second concern is how update transactions are actually processed at remote replicas. So far, we simply indicated that write operations are executed at all replicas. But once we consider that write operations are typical SQL statements, there are several practical issues to resolve. The second section is dedicated to update processing.

This chapter also discusses other important architectural issues such as replica transparency and replica discovery. The third section is devoted to this topic.

Finally, we discuss group communication systems as their use has become widespread in architecting database replication. The reason is that group communication systems help dealing with replica failure and recovery, as well as providing multicast primitives with ordering and reliability (atomicity) properties – which has proven to be helpful for enforcing 1-copy isolation, atomicity and durability. The fourth section provides an introduction to group communication systems and outlines how their functionality can be leveraged to architect database replication protocols.

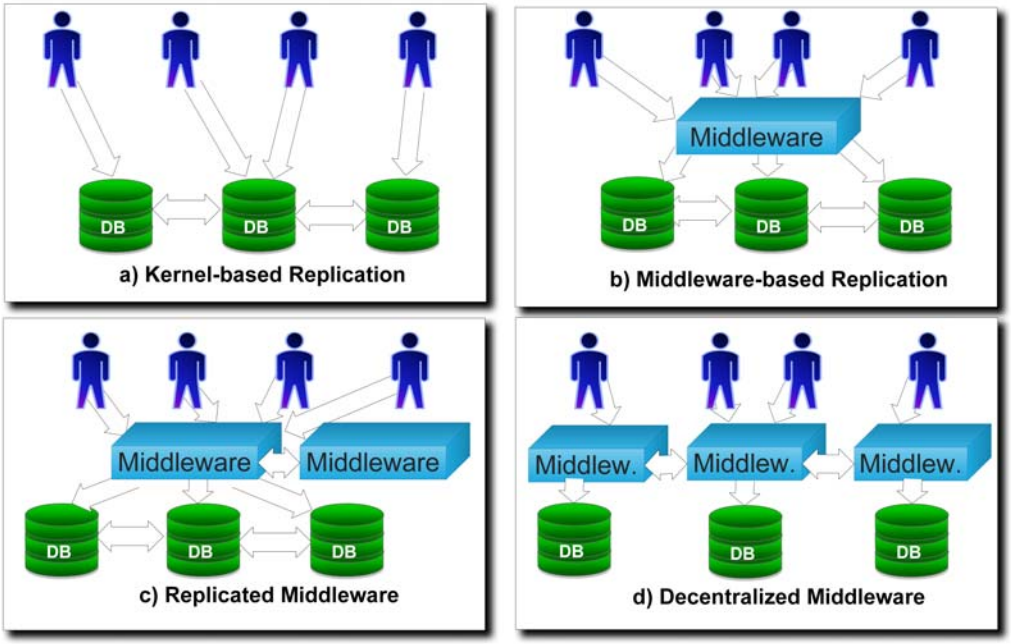

Figure 4.1: Alternative architecture

4.1 WHERE TO LOCATE THE REPLICATION LOGIC

4.1.1 KERNEL BASED ARCHITECTURE

The most natural location for the replication logic is the database kernel. Within the database kernel one has access to whatever functionality might be needed to replicate data and to integrate the replication logic with transactional processing. For example, it is quite easy to tightly couple replica control with the concurrency control mechanism. We refer to this approach as *kernel based* or *white-box database replication* since the database is like a white box with all its functionality visible to the replication module. This architecture is depicted in Figure 4.1.a. The database replicas coordinate with each other for replica control purposes. The clients typically connect only to one of the database replicas and only interact with this replica.

4.1.2 MIDDLEWARE BASED ARCHITECTURE

An alternative to kernel based replication is to encapsulate the replication logic into a separate component known as database replication middleware. The middleware is interposed between the database clients and the database replicas. Each database instance is a regular database without awareness of replication. From the database perspective, the middleware is simply a regular client.

The real clients send all their read, write, commit and abort requests to the middleware instead of the database. And the middleware is the one that takes care of coordinating transaction execution at the different database replicas, propagating updates, and ensuring that 1-copy equivalence is provided.

Centralized middleware. A replication middleware in its simplest form can adopt a *centralized middleware* approach (see Figure 4.1.b). In this approach, there is a single instance of the middleware between clients and database replicas. It receives all client requests, schedules them, and forwards them to the relevant database replicas. It receives the replies from the database replicas and returns them to the corresponding clients.

Replicated centralized middleware. The centralized middleware approach is simple, but unfortunately, it becomes a single point of failure. Thus, it fails to provide one of the main features replication is used for, namely availability. This problem can be avoided by replicating the middleware itself, purely for fault-tolerance purposes. Figure 4.1.c shows this architecture. One middleware is the master replica and all clients connect to this master, while the other is a backup middleware that only takes over if the master fails. We call this a *replicated centralized middleware*. As the middleware maintains some state (e.g., currently active transactions, some write information etc.), the backup middleware replica must receive state changes from the master middleware at appropriate synchronization points. There exist many different process replication techniques that are suitable to replicate the middleware such as active replication [Schneider, 1990]. However, in principle, the failover is similar to the failover of a database replica and we discuss this in more detail in Section 9.1.

Decentralized middleware. A third alternative lies in having a middleware instance collocated with each database replica resulting in a *decentralized middleware* approach as depicted in Figure 4.1.d. The middleware replica and the database replica together build one replication unit. The middleware replicas communicate with each other for coordination purposes. Clients typically connect to one of the middleware replicas.

This architecture has two advantages over a replicated centralized middleware. First, there is typically only one failover mechanism for the unit consisting of middleware and database replica. In contrast, a replicated centralized middleware needs to implement different failover mechanisms for the middleware and the database replicas.

Second, a decentralized middleware is more attractive in a wide area setting. If there is only one (master) middleware, all clients have to connect to the same middleware which might be far from the client, even if one database replica is close. Thus, all interaction between client and middleware crosses the wide area link. Similarly, the single middleware might be close to some but not all of the database replicas, leading to long-delay communication between middleware and database replicas. In contrast, with a decentralized middleware, a client can connect to the closest middleware replica. Both the communication between client and middleware replica, and middleware replica and database replica is thus local and fast. Only the communication between middleware replicas is across the wide area network.

4.1.3 KERNEL VS. MIDDLEWARE BASED REPLICATION

Advantages of kernel based replication. The major advantage of kernel based replication is that the replication module has full access to all the internals of the database system. Most importantly, it can be tightly coupled with the concurrency control mechanism. As shown in the algorithms of the previous chapter, concurrency control and replica control appear highly tangled, especially in eager approaches. In a kernel based approach that is quite easy to achieve. In contrast, a middleware approach does not have access to the concurrency control mechanism of the database system. Thus, many systems partially re-implement concurrency control at the middleware layer and might not be able to offer the degree of parallelism that is possible in kernel based systems. In particular, concurrency control in the database system is usually on a record basis, i.e., in locking based schemes, each record is individually locked just before it is accessed for the first time. In contrast, at the middleware we might not even know which records are accessed. Clients submit their requests typically in form of SQL statements and each individual statement can access many records. Thus, if the middleware does its own locking, it is typically on a coarse granularity such as tables. We also see shortly that executing write operations at remote replicas can be better optimized in a kernel based approach.

A second advantage of kernel based replication is that clients remain directly connected with the database system. In contrast, middleware systems introduce a level of indirection, leading to more messages in the system. However, middleware and database replica typically remain in the same local area network where this additional message overhead has relatively little impact.

Disadvantages of kernel based replication. The blessing of kernel based replication, namely the large optimization potential, is also its curse. If replica control is too tightly interwoven with concurrency control or the processing of records, any change in their implementation will likely directly affect the replication module. Maintenance of the code becomes a problem. In contrast, middleware approaches are forced to separate the concerns as the replication module can only interact with the database through a well-defined interface. Thus, implementation changes within the database are unlikely to affect the middleware system.

An important hurdle for kernel based replication, many times unsolvable, is the requirement to access the source code of the database. In case of commercial database systems, only the vendor itself can implement the replication solution as no one else has access to the source code. For open source databases, such as PostgreSQL [PostgreSQL, 2007], there exist, in fact, several replication solutions. However, the internals of the database system are usually extremely complex, and modifying and extending the code needs considerable experience with the underlying system. In contrast, middleware systems are developed independently, possibly from third parties, and can decide on their own internal structure.

Finally, a kernel based solution is confined to a single database system. In contrast, a middleware system can possibly use different database systems, and thus, can implement a replication solution across a heterogeneous environment.

4.1.4 BLACK VS. GREY BOX MIDDLEWARE

One of the major disadvantages of a middleware approach is that it has a very restricted interface to the database replicas and it is difficult to take advantage of the functionality provided by the database. Systems that use only off-the-shelf database interfaces without any replication support from the database represent a *black-box replication* solution, as they see the database as a black box. However, in some cases, implementing some extra functionality within the kernel, and exposing or exporting this functionality through appropriate interfaces to the outside, can come a long way to help the middleware perform its replication tasks. We will see later several examples where such functionality is useful. In this case, we refer to a *gray-box* approach as the database replica is not completely replication oblivious.

4.2 PROCESSING OF WRITE OPERATIONS

Write operations have to be executed at all replicas, and processing updates is the main overhead of replication. Therefore, it should be done as efficient as possible. So far, we have simply indicated that the write operations of update transactions are executed at both the local and the remote replicas. In relational database systems, a write operation is typically an SQL update, insert or delete statement. Executing a write operation means parsing the statement, determining the number of tuples affected and then perform the modification/deletion/insertion. If all replicas indeed perform all of these tasks, then we call this *symmetric update processing*. However, this can quickly waste valuable resources. Second, it requires that execution of these operations is completely deterministic. An example of a non-deterministic operation is an update that sets an attribute to the current time. As operations do not execute at exactly the same time at different replicas, symmetric update processing would allow data copies to diverge.

An alternative to symmetric update processing is what we call *asymmetric update processing*. When the transaction submits a write operation on data item x to a replica R^A, R^A does not immediately forward the write operation to the others even if the approach is eager. Instead, it first executes the operation locally, and then bundles the changes into a single message. That is, the identifier (e.g., the primary key) and the after-image of each updated record are collected. This information is sent to the other replicas, which can quickly find the affected records through their identifiers, and apply the updates directly. Applying these changes is much faster than executing the original SQL statement. In the following, we refer to the extracted changes as writeset, in order to indicate that this is different from the original write operations.

Note that the concept of symmetric vs. asymmetric update processing is orthogonal to whether the replica control protocol is eager or lazy, primary copy or update anywhere, as most replica control protocols can use both of the update processing mechanisms.

The question is how the writeset information can be extracted. Many different approaches have been proposed.

Triggers. As a first option, the writeset can be obtained using triggers, a functionality widely available in most database systems. Whenever an insert/update/delete occurs, a trigger is fired that captures the particular changes. The trigger could write the necessary information into an extra table where it can be later retrieved. Triggers are heavily used both in kernel and middleware based approaches. Many systems have internal trigger mechanisms for various purposes that can be reused for writeset collection in kernel based replication. Most database systems also provide triggers at their interface, making them accessible to a middleware system.

Log mining. Another possibility is to use the log mining facility available in some database systems. Databases usually write for each update performed the after-images of the affected records into a log. This is done for recovery purposes. Thus, writeset information can be easily extracted from the log. This appears particularly appealing for kernel based replication as they have direct access to the log. Some systems also export log access via interfaces. However, in this case, this is mostly, if not all, only after the transaction has committed. Therefore, middleware based approaches can only exploit this mechanism in case they use lazy replication where writesets are sent after commit.

Writeset extraction service. Instead of awkwardly extracting writeset information through triggers or the log, the most efficient approach is to create them as the records are accessed. Just as logging creates before- and after-images while the updates take place, so could the writeset be built on the fly, being optimized for transfer over the network and application at the remote replicas. Clearly, such writeset creation can only be done in the database kernel. However, it would not be difficult to expose the functionality to the outside. And here comes the gray-box approach into play. Let us have a closer look at the options.

- *SQL writeset.* The writeset could be created as a list of simple SQL update, insert and delete statements where each statement writes exactly one record. Thus, instead of having complex SQL WHERE clauses, the reference is always only to the primary key. If the database allows the middleware to retrieve such a writeset, then applying the writeset at remote replicas simply means sending the individual SQL statements. Application should be faster than the original write operation as access to the records can always be through the primary key index.
- *Opaque binary writeset.* Another possibility is to obtain the writeset in binary form. This is more efficient, since it does not require transforming the internal record format into SQL. It requires, however, a complementary service to apply the binary writeset at remote replicas. This has also the additional advantage that applying the writeset avoids the overhead of SQL processing. The disadvantage is that the middleware cannot read the writeset. As we outlined before, middleware systems often re-implement part of the concurrency control. For that purpose, it might be helpful for the middleware to know exactly the records that were actually changed by an operation. Also, sometimes the middleware might want to do some additional processing, in which case it would also need the details of the changes.

- *Transparent binary writeset.* To avoid the disadvantages of the opaque binary writeset, the database could also provide an interface to access the content of the writeset. Thus, record identifiers could be extracted, which can be used, e.g., for conflict detection.

4.3 PARTIAL REPLICATION

The previous section discussed the possible performance gain of asymmetric update processing by avoiding to execute the full write operations everywhere. Nevertheless, although applying the changes might be faster, they still have to be applied at all copies. One way to further reduce the write overhead is to reduce the number of copies per data item. Using partial replication, each data item has only copies on some but not all of the nodes[1]. If a node does not host a copy of a data item then it does not need to apply the changes, leaving more capacity for doing other work. Therefore, the idea is to create just enough copies of a data item so that the read load on this data item can be nicely distributed among all copies.

However, partial replication has its own difficulties. In *pure partial replication* each node has only copies of a subset of the data items but no node contains a full copy of the database. In this case, it is possible that a transaction accesses a set of data items such that no node has copies of all of them. Thus, distributed transaction processing needs to be supported. This is complex as it requires exact knowledge of where data items reside. Also, the typical client interface for relational databases are SQL based, and SQL select statements can translate to read operations on many data items. All of these data items must reside on one node to perform the operation, but often it is not possible in advance to determine which set is actually going to be accessed. This reduces the flexibility of how the database can be distributed. For instance, it might require that all records of a table be collocated on a node.

Furthermore, update propagation is a further challenge. For instance, assume node N^A has copies of data items x and y, and node N^B has copies of y and z. Assume a transactions starts at N^A and reads x and then writes y. Later it submits a read (e.g., SQL select) operation that accesses y and z. This read operation can only be served by N^B, but this node might not yet have the current version of y, e.g., because the writeset is only propagated at the end of transaction. Thus, replica control becomes more complex.

There are two ways to avoid distributed queries. One is *hybrid partial replication*. In this approach, a set of nodes are full replicas that have a copy of the entire database, and another set of nodes are partial replicas containing only a fraction of the database. Hybrid partial replication analyzes each SQL statement and decides where to execute it. If there is a partial replica that contains all data items that might be accessed, then the statement can be executed at this partial replica. Otherwise, the statement must be executed at a full replica. The main issue with this approach is that full replicas have to apply all write operations. As the number of transactions increases, the full replicas will eventually saturate, building the bottleneck in the system. Figure 4.2 depicts pure and hybrid partial replication.

[1] In case of partial replication, we prefer to use the term “node" instead of “replica".

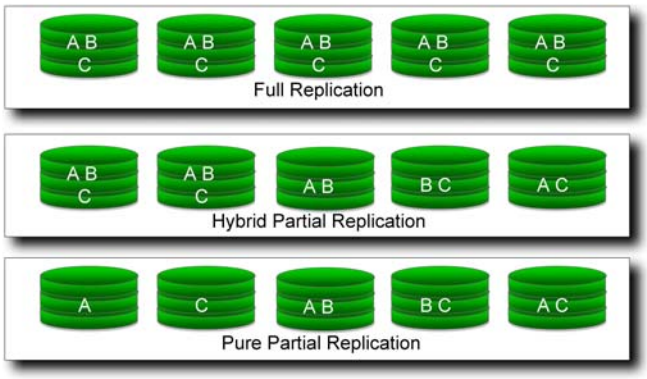

Figure 4.2: Full, hybrid, and pure partial replication

The other technique to avoid distributed transactions is to exploit a priori knowledge of transactions. If the set of transaction types and the data items each transaction might access are known in advance, then the database can be distributed such that for each transaction there is at least one partial replica that has all data items that will be accessed by the transaction. Therefore, each transaction can execute fully on one node.

In any case, deciding on where to put which data items in order to optimally exploit the resources of all nodes and to not create any bottlenecks in the system, is a challenging task and requires a good understanding of the application requirements. Furthermore, replica control can become more complicated in some cases. The replica control protocols we present in the following chapters all assume full replication although some might be easily extensible to partial replication.

4.4 OTHER ISSUES

Replication transparency. A very desirable feature of any replication solution is replication transparency. It allows database applications to be kept unmodified. This is very important, since otherwise every database application would need to be modified to adjust for replication. That would make any replication solution hard to apply and maintain. From an architectural point of view replication transparency means that the replicated system has to provide exactly the same interface to the database application as a non-replicated database system. Databases are today accessed by means of database connectivity components such as JDBC and ODBC. These components are split into two modules, a client-side component and a server-side component. Clients connect to the client-side component that provides a standardized interface. Thus, whether middleware or kernel based replication is used, the application should be able to load a corresponding client-side component that offers the standard interface. The client-side component can now implement some of the replication

functionality, hiding it from the application. In particular, two main functionalities that are usually implemented in the client-side connectivity component are replica discovery and failover.

Replica discovery. In most of the architectures that we have discussed in Section 4.1, there is no longer a unique access point for clients. The only exception is the centralized middleware where there is a single middleware where all clients connect to. Otherwise, the middleware is replicated or decentralized, or the database replicas are directly accessed. Each of them has its own unique address, such as an IP address. Furthermore, this set is also dynamic, as replicas might fail and new replicas are added. This is in contrast to a non-replicated system where a client application connects to a well-specified single address. Therefore, the replicated system has to provide a mechanism to detect replicas. Let us have a look at two mechanisms to detect replicas.

- *IP-multicast-based replica discovery.* One possibility for replica discovery is to resort to IP-multicast. All replicas subscribe to a specific IP multicast address that is specifically used to detect replicas. When a client wants to connect to the replicated database the connectivity component at the client IP-multicasts a replica discovery message. Replicas receive the message and reply to it, typically with their real IP address, and possibly other information such as load and configuration information. The connectivity component can then select a replica to connect to according to the replies it received. In particular, in a primary approach it has to connect to the primary if it wants to submit update transactions. IP-multicast-based replica discovery is only possible if IP multicast is supported. Thus, it is typically restricted to cluster replication within a local area network.

- *Directory-based replica discovery.* An alternative is to rely on a directory service. In this case, a directory node with a well-known IP address keeps a directory of all available replicas and their IP addresses. The node monitors the current set of available replicas and updates the directory content regularly. Thus, the client connectivity component can request the directory information from the directory node and then connect to one of the available replicas. Of course, the directory node now becomes a single point of failure and might require replication by itself for fault-tolerance purposes.

Failover. Failover functionality is somewhat related to replica discovery. When a replica fails, the clients that are connected to this replica have to reconnect to another replica. This is part of the failover procedure. Typically, when a node fails, the client loses the connection and receives a failure exception upon its next request. The client connectivity component has to transparently catch this failure, find a new replica and reconnect. To find a new replica, one of the discovery mechanisms above can be used. The client component could also have stored the available replicas when it made its first connection. Alternatively, information about available replicas can be forwarded regularly to clients by piggybacking it on standard messages transmitted to the client.

4.5 GROUP COMMUNICATION AS BUILDING BLOCK

A replicated database has to deal with many issues related to distribution: failure detection, coordination among replicas, reliable and ordered message exchange, etc. Providing such functionality is hard, especially in the advent of failures. But this challenge is not unique to replication, and a large body of research has proposed general-purpose solutions for these issues. In particular, group communication is a paradigm that provides most of these functionalities. This means that group communication is an excellent building block to simplify the architecture and protocols of a replicated database. In fact, over the last decade many replication solutions have been proposed that rely on the functionality of group communication systems to simplify the tasks at the replication layer. In the next sections, we give an overview of the key functionalities provided by group communication. Then, we shortly outline how replication can exploit these properties. More details will be given in the following chapters when we describe individual replication solutions in more detail.

4.5.1 GROUP COMMUNICATION AND RELIABLE MULTICAST

A group communication system is a communication middleware that provides an advanced interface to the application. It is typically implemented as a layer between the standard point-to-point communication (e.g., UDP/IP) and the application. Application processes are distributed over a set of nodes and communicate and interact via the group communication interface provided by the group communication layer. The group communication paradigm provides the notion of process group. The application processes build a process group that cooperates together for a given task, for instance, to replicate a database. The group communication layer provides two main functionalities: group membership and multicast. Furthermore, the two features are interrelated through the notion of virtual synchrony.

Membership. Group membership provides the notion of views, where a view is the current list of connected and alive processes in the group. Application processes can join/leave the group by submitting a join/leave request. Furthermore, the group communication layer detects process failures and automatically removes the failed processes from the group. Every time that there is a change in the membership due to a join/leave or a failure, each available application process receives a *view change message* informing it about the membership of the new view.

Multicast. Reliable multicast allows sending a message to all available group members. When multicasting a message one can specify reliability and ordering properties. This requires additional coordination within the group communication layer. Thus, when the group communication layer receives a message from the underlying network, it first ensures that the reliability and ordering properties are guaranteed. Only then the group communication layer *delivers* the message to the application layer.

Reliability. In point-to-point communication, reliability means that a message will be eventually received by the receiver as long as both sender and receiver remain available. When a message is

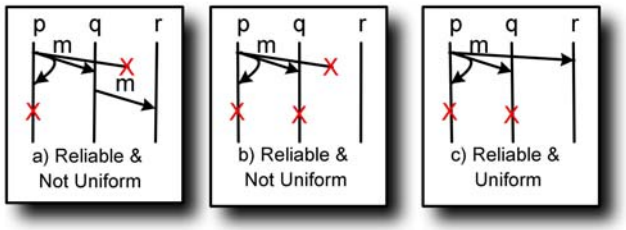

Figure 4.3: Reliable and uniform multicast examples

multicast to a group, we have many receivers. Thus, the concept of reliability becomes more complex. We can distinguish two levels of reliability, and their differences are subtle but important. They relate to faulty processes. For that, we have to first define what that means. We say that a process is *correct* if it is available, up and running, at least for the time period under observation. Otherwise, the process is *faulty*. Faulty means it crashes sometime during the execution of the message exchange. We only consider simple failures where a process simply stops working.

With this definition, *reliable* multicast guarantees that once a message is delivered to one correct process, it will be delivered to all correct processes. As a result, the same set of messages is delivered to all correct processes.

Let us have a short look at an example as depicted in Figure 4.3.a. There are three processes p, q and r building a group. While not depicted in the figure, each process has a group communication and an application layer. Assume the application layer of process p multicasts a message m to the group. Internally, the communication layer of p IP-multicasts m to the group or sends the message individually to the other processes using UDP. Since IP-multicast and UDP are unreliable, it might be possible that the group communication layer at p and q receive the message and deliver it to their application layers, but the message does not arrive at r. Now assume that p fails. We do not care whether the communication layer at p has delivered the message to its application, because reliable multicast does not care about faulty processes. However, since q is correct and the message was delivered, the message must also be delivered at r since r is also correct. Therefore, reliable multicast requires a resubmission mechanism so that the group communication layer at q can forward the message to r. This means that the group communication layers at correct processes have to keep track of the delivered messages and might need to forward them to other correct processes if failures occur.

An important aspect of reliable multicast is that it does not restrict what happens at faulty processes. Obviously, a message that is delivered by correct processes might not be delivered at a faulty process. But it might also be possible that a message is delivered at a faulty process while it is not delivered at the correct processes. Figure 4.3.b shows an example where p sends the message to

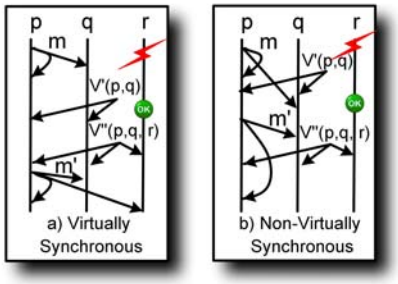

Figure 4.4: Virtual synchrony

q where it is delivered and then both p and q fail while the message never arrives at r and therefore, is not delivered at this correct process. This is allowed with reliable multicast.

Uniform reliable multicast. Some applications also want to have control over what happens at faulty processes, and we will see later that this is true for database replication. Uniform reliable multicast provides stronger guarantees than reliable multicast. It guarantees that whenever a message is delivered at any process (independently of whether the process is correct or it is faulty and crashes shortly after delivery), then it is delivered at all correct processes. In this case, the execution of Figure 4.3.b is impossible, as a delivery at p or q implies a delivery at correct process r. It basically means that when the group communication layer of q receives the message from the network it does not immediately deliver it to the application. Instead, it waits until it knows that the communication layer of r has received it as well. Only then, it delivers the message. If it fails before receiving this confirmation from r, it is guaranteed that the message was not delivered to the application. That is, the fundamental property of uniform reliable multicast is that the faulty processes deliver a subset of the messages delivered at correct processes. Figures 4.3.a and b show cases that are prevented by uniformity. Figure 4.3.c shows the only case that would be allowed under uniformity in addition to no process delivering the message m.

Message ordering. Group communication systems typically provide three levels of message ordering: FIFO, causal and total. With FIFO, ordering messages from the same sender are delivered in sending order to each group member. Causal ordering is the transitive extension of FIFO via casual dependencies. Total ordering guarantees that all messages are delivered in the same order to all group members independently of who sent them.

Virtual synchrony. Virtual synchrony relates message delivery with view change events. Informally, it guarantees that each application perceives view changes at the same virtual time in regard to message delivery. More formally, virtual synchrony guarantees that two processes that transit from a

view V to a consecutive view V' get the same set of messages delivered while being members of V. In the examples depicted in Figure 4.4, the current view V consists of p, q and r, and then r fails. Then, a view change message is delivered at p and q informing them that the new view V' consists of only p and q. In Figure 4.4.a a message m is delivered at p just before the view change V'. Under virtual synchrony, it is guaranteed that m is also delivered at q before the delivery of V'. Assume now that r rejoins. A new view change V'' is delivered at all three processes containing again the three processes. Under virtual synchrony, if a message m' is delivered at p after the delivery of V'' then m' is also delivered at q and r after delivery of V'' (unless they fail, of course). However, without virtual synchrony the scenario in Figure 4.4.b might happen. Message m is delivered at p but not at q before the delivery of V'. This violates virtual synchrony. The same happens with m' that is delivered at p before V'' and at q after V''.

4.5.2 SIMPLIFYING REPLICATION WITH GROUP COMMUNICATION

There exist many eager replication solutions that take advantage of some of the group communication primitives. We briefly outline how these primitives can be exploited to support replication semantics.

Exploiting group membership. One of the functionalities required by replication protocols is membership. That is, a replication protocol should keep track of which replicas are available and connected. This functionality can be delegated to the group communication system by letting each replica be member of a replication group. In this way a failure of the node holding a replica is automatically detected by the group communication system. The replica could be a database replica in kernel based replication or a middleware replica if a replicated centralized or a decentralized middleware architecture is used. Basically, the group communication system detects the failure and produces a view change that is delivered to each available member of the group. This triggers the failover procedure at the replication layer. Group membership is not only useful for detecting failures. It is also useful to keep track of new replicas joining the system, either after a recovery of a failure or when new replicas join the system.

Exploiting ordered multicast. Using an ordered multicast can simplify the task of replica control, and thus, helps to achieve 1-copy-serializablity or other 1-copy-isolation levels. Let us take as a simple example the eager update anywhere protocol of Figure 3.1 from Section 3.1. In Section 3.1.4, we outlined how this protocol can produce distributed deadlocks. The problem is that write operations can be submitted concurrently to different replicas, these replicas acquire the locks locally and then send the requests to the other replicas where they have to wait for each other. Such distributed deadlocks are difficult to detect. However, if a replica multicasts a write operation in total order, then the write operations are delivered to all replicas in the same total order. Now each replica only has to request locks in the order in which the write operations are delivered. As a result, all replicas acquire locks in the same global order, and distributed deadlocks can be avoided allowing only local deadlocks that are easier and cheaper to detect and resolve. We will see in later sections various protocols that exploit this total order to determine the global serialization order. In fact, the use of

total order multicast often allows each replica to decide locally the serialization order of transactions, and whether a transaction can commit or has to abort because of conflicts.

Exploiting uniform reliable multicast. Uniform reliable multicast can be used to guarantee 1-copy-atomicity despite failures. In Section 3.1 we have argued that eager protocols have to be enhanced with an agreement protocol, such as 2-phase-commit, to guarantee that all replicas decide on the same outcome of a transaction in failure cases. Instead, in the protocol of Figure 3.1, it is actually enough that the local replica multicasts the commit message to all replicas using uniform reliable multicast. If the message is delivered at any replica and the transaction committed, it is guaranteed that the message is delivered at all correct replicas, and thus, the transaction committed. In particular, once a message is delivered to the replica that sent it, and this local replica commits the transaction and informs the client about the commit, all correct replicas will commit, too. Therefore, even if the local replica fails shortly after the commit, it is guaranteed that the available system has the transaction committed.

Uniformity also simplifies recovery due to the guarantee that a failed replica has only committed a subset of the transactions that were committed by available replicas. Thus, recovery only has to send the missing transactions to the recovering replica, but there is no need to reconcile transactions that only committed at the failed replica.

One has to note, however, that implementing uniform reliable multicast requires by itself some agreement protocol. The group communication layer of a replica cannot simply deliver a message once it receives it but has to delay the delivery until it knows that the message has arrived at all other replicas. However, uniform reliable multicast is typically much faster than 2-phase commit because it is implemented in the communication layer, and more importantly, it does not use any logging to persistent storage (i.e., disk).

Exploiting virtual synchrony. Virtual synchrony can come in handy at the time when a new node or a previously failed replica joins the system, and thus, can help with the task of 1-copy-durability. The joining replica needs the current state of the database. Typically, one of the available replicas transfers it. However, such a transfer can take a long time, and transaction processing typically continues during this transfer. The joining node may not miss the updates of any of these transactions, i.e., it either receives the updates of these transactions as part of the database transfer, or it has to execute these transactions by itself after the transfer is complete. Virtual synchrony can help to determine the point at which the joining replica switches to processing transactions by itself. For instance, the new replica could join the group shortly before the transfer is complete. A view change is delivered to all replicas indicating that the new replica has become member of the group. The replicas also know that all messages that are delivered after the view change are also delivered to the new replica. Therefore, the replica performing the transfer could transfer all changes related to messages delivered before the view change but let the new replica process all the messages that are delivered after the view change.

4.6 RELATED WORK

Basically, all of the early work on database replication and also the first approaches that appeared in the late 90s assume kernel based replication. Concurrency control at each replica is tightly coupled with replica control, providing one coherent replication solution. The replication solution is also tightly coupled with the other components of a database system. Most of these solutions are evaluated based on simulation, and thus, are not concerned with how these ideas can be truly implemented, either within a concrete database kernel or whether it would be actually possible to implement them in a middleware layer. Also, how write operations are actually executed at remote replicas is not clearly described in most of this early work.

A first concrete prototype implementation [Kemme and Alonso, 2000b] integrates an eager protocol based on group communication into the PostgreSQL engine. It directly extends the existing locking-based concurrency control mechanism and uses asymmetric update processing, taking advantage of the internal structure of records. A follow-up work [Wu and Kemme, 2005] moves the implementation to a newer version of PostgreSQL that is based on snapshot isolation. Manassiev et al. [2006] present a replication solution within the MySQL server. Little other research work on real kernel based implementations exists due to the complexity of database kernels and lack of publicly available database engines. In contrast, commercial systems all provide various forms of replication, typically embedded within the database kernel.

The seminal approach to middleware based database replication was Middle-R Jiménez-Peris et al. [2002b]; Patiño-Martínez et al. [2005]. Many middleware based approaches have followed [Amir and Tutu, 2002; Amza et al., 2003a,b; Cecchet et al., 2004; Elnikety et al., 2006; Lin et al., 2005; Röhm et al., 2002]. While most of the approaches use a black-box approach, Middle-R [Jiménez-Peris et al., 2002b; Patiño-Martínez et al., 2005] follows the gray-box approach as PosgreSQL is extended with two services to extract and apply opaque binary writesets. Another gray-box approach is Tashkent [Elnikety et al., 2006], which proposes to expose an interface to tag transactions with ordering that should be enforced by the database. In this way, writesets can be committed in parallel without the risk of the underlying database changing the ordering established by the middleware. Furthermore, Correia et al. [2007] and Salas et al. [2006] present reflective database architectures suitable to build gray-box replication. Reflection exposes database behavior to the middleware and, at the same time, permits the middleware to intercept the transaction processing within the database and add/change the behavior of the database. The C-JDBC [Cecchet et al., 2004] implementation has evolved into a commercial product.

Many black-box approaches use symmetric update processing, due to the difficulty of extracting the writeset. Exceptions are Ganymed [Plattner and Alonso, 2004; Plattner et al., 2006a] and DBFarm [Plattner et al., 2006b] that use triggers to extract writesets. The cost of writeset extraction and application has been analyzed in Salas et al. [2006].

Most middleware approaches are centralized. Basically, all decentralized approaches, whether kernel or middleware based, use group communication for communication among the replicas [Amir and Tutu, 2002; Kemme and Alonso, 2000b; Lin et al., 2005; Serrano et al., 2008;

Wu and Kemme, 2005]. Tashkent [Elnikety et al., 2006] uses a hybrid architecture in which local middleware instances are collocated with each database replica, and clients interact directly with these replicas. However, there is a centralized certifier with which all middleware replicas communicate for concurrency control purposes. This certifier is replicated for availability purposes. Another possibility for a hybrid approach would be to let clients connect to a centralized middleware that communicates with middleware instances collocated with each database replica.

Cecchet et al. [2008] discuss existing research attempts to middleware based replication and analyze what is needed to make them work in a real industrial setting. The main criteria are performance, availability and administration.

Most partial replication protocols offer 1-copy-serializability [Fritzke Jr and Ingels, 2001; Holliday et al., 2002; Pacitti et al., 2005; Schiper et al., 2006; Sousa et al., 2001]. Fritzke Jr and Ingels [2001] use one total order multicast for every read operation and one for the writesets. Schiper et al. [2006] introduce partial replication algorithms based on group communication. In Holliday et al. [2002], one protocol requires all data to be accessed by a transaction to reside on one node, the other creates temporary copies for data items that are not locally replicated. Pacitti et al. [2005] describe a lazy replication protocol that allows both update anywhere and primary copy. For each table, a different mechanism can be used. It is a middleware based protocol that enforces the same total order of transactions at all replicas. There exists some work on partial replication and 1-copy-SI protocol [Serrano et al., 2007], its application to wide area networks [Serrano et al., 2008] and probabilistic analysis of abort rates Bernabé-Gisbert et al. [2008]. A performance analysis of partial replication was done by Nicola and Jarke [2000], and the question of where to locate replicas was discussed by Wolfson et al. [1997].

One of the first group communication systems to be developed was the ISIS system [Birman et al., 1991]. Further well-known systems are Totem [Moser et al., 1996], Horus [van Renesse et al., 1996], Ensemble [Hayden, 1998], Spread [Spread, 2007], Appia [Miranda et al., 2001], and JGroups [JGroups], with the last being frequently deployed in many open-source projects. A survey on group communication properties is provided by Chockler et al. [2001].