MASTER-SLAVE REPLICATION, FAILOVER AND DISTRIBUTED RECOVERY IN POSTGRESQL DATABASE

By

Mabrouk Chouk

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
MCGILL UNIVERSITY
MONTREAL, QUEBEC
JUNE 2003

© Copyright by Mabrouk Chouk, 2003

MCGILL UNIVERSITY

DEPARTMENT OF

COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**Master-Slave Replication, Failover and Distributed Recovery in PostgreSQL Database**" by **Mabrouk Chouk** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: June 2003

Supervisor: _____
Dr. Bettina Kemme

Readers: _____

_____

_____

To my family.

Table of Contents

Resumé

Les sujets traités dans cette thèse sont la duplication (replication) synchrone et de type maître esclave (master-slave) de bases de données, la récupération-après-échec (failover) du site-maître et le rétablissement-distribué (distributed recovery) d'un groupe de serveurs de bases de données.

Notre solution de duplication permet la mise-à-jour (update) seulement sur le site-maître. Cependant, les requêtes sont permises sur tous les sites. La mise-à-jour des transactions est envoyée à tous les sites avant d'être commises (committed) sur le site-maître. Pour ce faire, on exploite les sémantiques des systèmes de communication de groupe. Dans ce travail, on garantit toujours l'existence d'un site-maître dans le groupe, et cela est grâce à la récupération-après-échec dynamique du site-maître.

Notre solution de rétablissement distribué assure la cohérence des bases de données de tous les sites. Tout site introduit dans le groupe suivra un protocole où il va chercher toutes les transactions qui ont été traitées avant son introduction, ou pendant qu'il était en panne. Un nouveau site est prêt à partager les tâches du système uniquement après avoir accompli avec succès la récupération-après-échec et le mécanisme de rétablissement-distribué. La base de données utilisée dans notre travail est PostgreSQL.

Abstract

The subject of this thesis is synchronous master-slave database replication, failover and distributed recovery for a cluster of database servers.

Our replication solution allows updates on the master, and queries on all sites. Updates of transactions are sent to all sites before they are committed on the master. To do this, we take advantage of the semantics of group communication systems. With failover, there is always a master in the group. Distributed recovery ensures that a site that has been introduced in the system brings its own copy of the database to be consistent with the rest of the other databases. It does this by getting all the transactions it missed from another site while it was not in the group. It is only after having successfully carried out failover and recovery mechanisms that a new site is ready to share in the load of the system.

Acknowledgements

There are a number of people that helped me realize this work, some with technical expertise and some with 'not-so-technical' support, yet of paramount importance.

First and foremost, I would like to extend my deepest gratitude to my supervisor, Dr. Bettina Kemme, for her constant stream of ideas that helped me finish this work. She never stopped impressing me with her intelligence and her expertise. I am also very grateful for her patience and motivation. I especially like to thank her for encouraging me in those times where I felt that I was not capable of finishing this work. Special thanks go to my research group members, namely Huaigu Wu, Yisheng Liu, Jiafeng Wu, Qifang Zheng, Xueli Li and Yi Lin. We had numerous meetings where lots of concepts and ideas, around my work and theirs, were discussed. I have learned tremendously in those meetings.

Other people greatly affected my motivation and enthusiasm to see this thesis come to a successful conclusion. These people did not help me technically, but provided me with endless encouraging and support, although without knowing it sometimes. In particular, I would like to extend my admiration to my mother Salha Ouaja. She is the most patient, warm and caring individual I have known. Deep thanks also to Khaled, Bannour, Dalila, Naziha, Naima and Fethi.

Special thanks go to my friends and relatives. They contributed to the successful finishing of this work. In particular, I would like to thank Khalil, Hatem, Akram, Anis, Slim and Nouri. Also, I would like to thank Mathieu, Vahid, Maud, Jean Marie-eve, Caroline, Denisa and Ayako.

Chapter 1

Introduction

Over the last decade or so, we have witnessed tremendous growth in distributed database systems. Led by the explosive growth of computer processing power and storage, distributed database systems became widely spread. They have replaced the traditional central and legacy databases. A key to this paradigm shift is the concept of *database replication*. Database replication is the process of maintaining a defined set of data in more than one location. Each copy of the database is called a *replica*, and each may contain the full or a partial set of the complete set of data. The replicas usually reside on different machines, interconnected by a local area network or a wide area network. An important issue in database replication is *data consistency*, that is, that the different replicas have the same value despite updates. It requires to propagate changes performed on one replica to the other replicas. If updates are allowed to be performed on any copy in the system, concurrent updates on different replicas have to be synchronized.

Database replication can be achieved using a *master-slave approach*, where the master is the coordinator of replication and the other sites are the slaves. Transactions that do change data are applied only on the master, however transactions that don't change data can be applied on all sites. The master has to send the updates of the transactions to the slaves, which have to apply them to their local copy of the database when they receive them.

Because of the important role of the master in this work, mechanisms should be put in place to ensure that there is always a master up and running. *Failover* in a distributed system refers to the dynamic replacement of a new master if the old one has failed.

Failover is motivated by the need for system reliability and availability. The importance of these motivations vary widely from system to another. In some systems, they are of critical importance, and failure to ensure them may result in catastrophic results. Examples include aircraft flight-control systems, hospital patient monitoring systems, and financial on-line transactional applications. These applications require the underlying computer systems to continue providing their services even if the system endures hardware or software failures. Up to very recently, organizations resorted to custom designing their fault tolerant systems to ensure high reliability and availability. However, as it became known, customized systems are invariably very costly, in terms of building and maintaining. They also do not track technology trends.

In a distributed environment, databases can fail and recover. When a database in a replicated database system fails, the database system has to be able to isolate the failed server from the rest of the group. The process of excluding the failed site should have little or no effect on the behavior of the system. When this site is ready to rejoin the group, mechanisms should be in place to re-introduce the site back into the group without interruption to the database system. There are two distinct types of recovery mechanisms, namely *central recovery* and *distributed recovery*.

In central recovery, a database reverts to the last consistent state just prior to the failure. This is done as follows: For each transaction executed prior to the failure either all or none of the changes of the transaction are reflected in the database state. Transactions for which the user received the commit before the crash are guaranteed to be reflected in the database state, and the changes of an aborted transaction are guaranteed not to be reflected in the database.

Distributed recovery is the process by which the recovering site acquires the transactions it has missed while it was down. It is the process that brings the *recovering site* to have the same data as the sites that are up and running. Distributed recovery is different from cental recovery. In central recovery, a database brings its copy to a consistent state, regardless if other sites are present in the group. In distributed recovery on the other hand, the database has to bring its own copy to be identical to the copies of the other sites. Central and distributed recovery are *both necessary and complementary* to completely recover a particular site.

The objective of this thesis is to provide a practical solution to master-slave replication, failover and distributed recovery. In the first step, we provide protocols to the above concepts. We base our protocols on recent work in the research field of distributed databases. We then provide a practical solution in the form of an implementation to test the feasibility and correctness of our protocols. The implementation part is carried out on the PostgreSQL database, version 7.2. We have adopted the name Postgres-RFR to refer to the modified version of this database that reflects the addition of the three new modules: replication, failover and distributed recovery. This implementation provides helpful insights into the practical issues of turning theoretical protocols into a working system. We were able to adjust our approach to the specifics of PostgreSQL without any conceptual changes to the database. One important aspect in our work has been to find a modular design that minimizes the changes that are necessary to the original system. In fact, the majority of our implementation could be added to PostgreSQL as separate modules and only few modules of PostgreSQL had to be changed. With this, we believe that our approach can be integrated into a variety of different systems.

A key to the correctness of our solutions is the underlying group communication system. To handle the large amount of communication between the database servers, we used the *Spread Group Communication System.* Spread provides valuable services in the form of delivery guarantees, such as reliable message delivery, and group memberships. These services are of great importance to ensure the feasibility and correctness of our protocols. Using Spread, we were able to focus on the correctness

of our protocols and were able to abstract the underlying communication semantics.

The present thesis is structured as follows: Chapter 2 outlines important database concepts that are pertinent to replication, recovery and failover. Chapter 3 describes the modules of PostgreSQL database that will be modified to accommodate our implementation. Chapter 4 gives an overview of the group communication system used. Chapter 5 gives details about the replication solution. Chapter 6 describes failover mechanisms in detail. Chapter 7 is concerned about distributed recovery. Chapter 8 gives final remarks and concludes this report.

Chapter 2

Overview of Database Concepts

In this chapter, we introduce important database concepts. These concepts will lay the foundation for our work in the following chapters. We will pay particular attention to the requirements of replication and distributed recovery.

2.1 Transactions

A transaction is a series, or list, of *actions*. The actions that can be executed by a transaction include *reads* and *writes* of database objects. A transactions can also be defined as a set of actions that are *partially ordered*, that is the relative order of some of the actions may not be important. In addition to reading and writing, each transaction must specify as its final action either a *commit* (i.e. completes successfully) or *abort* (i.e. terminate and undo all the actions carried out thus far). Transactions may also begin their actions with a *begin* (i.e. the point where the transaction starts). An *end* is similar to a commit, and a *rollback* is similar to abort.

Transactions must also satisfy four important properties. These properties are sometimes referred to as the ACID properties.

1. **Atomicity:** operations of all transactions occur as an un-dividable unit.

2. **Consistency:** transactions have to be executed on a consistent database, and will keep it that way.

3. **Isolation:** Execution of one transaction is isolated from that of another.

4. **Durability:** If a transaction commits, its effects persist, even in the case of system failure (not necessarily media failures).

2.2 Database Replication

Database replication is a well studied field [7] with many contributions from the research community [4] [5] [9] [15] [17] [18] [19] [20] [29] [30] , as well as from commercial research and development [14].

Replication is used for two purposes, namely to enhance system performance and fault tolerance. Reading local data and therefore balancing the load on the system usually improves performance. With replication, the data on available sites are accessible even in case of failures. However, replication mechanisms for fault-tolerance often lack performance and scalability. On the other hand, efficient and scalable replication mechanism often lack fault-tolerance. This means that often only one goal is achieved and the other is violated.

2.2.1 Database replication alternatives

Experts in the database industry classify database replication into four distinct categories depending on when replication occurs and where the updates are allowed to take place [14].

Transaction updates are replicated in two different ways: before the transaction commits or after. *Eager replication*, also called synchronous replication, is the process of replication that takes place before the transaction commits, whereas *lazy replication*, also called asynchronous replication, is the process of replication that takes place some time after the commit.

Replication could also be designed so that updates are allowed only on one replica, and read operations allowed on all sites. This is called *primary copy replication.* The database server where the updates take place is usually called the *master* or the coordinator, and the other database servers the *slaves.* Database replication can also be designed so that updates take place on every replica, in which case, the replication is denoted as *update-everywhere replication* or multi-master replication. Figure 2.1 summarizes these alternatives.

	Eager (Synchronous)	**Lazy (Asynchronous)**
Primary Copy (Master)	Updates allowed on one replica only Updates propagated before transaction commits	Updates allowed on one replica only Updates propagated after transaction commits
Update-Everywhere (Multi-master)	Updates allowed on all replicas Updates propagated before transaction commits	Updates allowed on all replicas Updates propagated after transaction commits

Figure 2.1: Replication Schemes

2.2.2 Basic replication strategies

One of the reasons for using replication is to increase database availability [7]. By storing data at multiple sites, the database can operate even if some sites have failed. In an ideal world where sites never fail, the database system can easily manage replicated data. It translates each data read operation to a read on one copy of that particular data, and translates a data write operation to a write on all copies of the data. This is the *Read-One-Write-All*, or ROWA approach to replication.

Unfortunately, sites can fail and recover. This poses a problem for the ROWA approach, because it requires that the database system processes each write operation by writing into all copies of that data, even if some have failed. If the database system were to adhere to this write all requirement, it would have to delay processing writing into data until all data copies are available.

Since there will be times when some copies of data are down, the database system will not be able to efficiently satisfy the ROWA write all requirement. For this reason,

there exists a more flexible approach to replication, namely *Read-One-Write-All-Available*, or ROWAA approach to replication. The risk with the ROWAA approach is that data copies might be out of date just after a site recovers from a failure. For this reason, transactions are prevented from reading copies from sites that have failed until those copies are brought up to date.

There exists another approach to replication, namely the *quorum oriented replication*. In this approach, each site is assigned a non-negative weight. Each site knows the weight of all other sites. A quorum is any set of sites with more than half the total weight of all sites. Only the one component which has a quorum of sites is allowed to process a transaction.

2.2.3 Replication in commercial systems

Various database replication solutions exist in the industry. The major companies that have replication products are Sybase, IBM, Oracle and Lotus Notes [1].

Sybase product is called Sybase Replication Server. It is used primarily to avoid server bottlenecks by moving data to the clients. To maintain performance, asynchronous, primary copy replication is used. The changes are propagated on a transaction basis, and after the transaction commits. Capture of changes is done using the log to minimize the impact on the running server. Replication takes place on a subscription basis, where sites subscribe to copies of data, and changes are propagated from the primary as soon as they occur [32].

IBM has a similar product, IBM Data Propagator. The goal of the replication mechanism is information warehousing and management, where complex views of the data are provided for decision-support. Replication is asynchronous, primary copy and there are no explicit mechanisms for updating. The IBM product uses a capture/apply mechanism where the replicas have to request the changes from the primary site to keep up to date. The system supports sophisticated features such as sophisticated data replication, sophisticated optimizations for data propagation and sophisticated views of the data [16].

Oracle product is called Oracle Symmetric Replication. The primary goal of replication here is flexibility. It tries to provide a platform that can be tailored to as many applications as possible. Asynchronous replication is the default but synchronous replication is also possible. It provides several approaches to replication and the user must select the most appropriate to the application. Changes to a copy are captured by triggers, which execute a remote procedure call (RPC) to a local queue and it inserts the changes in the queue. These changes take the form of an invocation to a stored procedure at the remote site. Queues follow a FIFO discipline and a 2PC is used to guarantee that the call makes it to the queue at the remote site [27].

Lotus Notes implements asynchronous lazy update everywhere replication in an epidemic environment, where updates are propagated after the transaction commits. Lotus allows to specify what to replicate to minimize overhead. It is a cooperative environment, where the goal is data distribution and sharing. Consistency is largely user defined and not enforced by the system [26].

2.2.4 Our work and current research

Our work is based on research in the field of database replication in general as outlined in 2.2, and specifically on the ideas presented in [18] and [20].

In [20], the authors develop and implement an eager update everywhere replication tool that provides performance and scalability. These are accomplished in three steps. First, a theoretical framework, including a suite of different replica control protocols is presented. In a second step, the feasibility of the approach is validated both by means of a simulation study and the integration of the approach into an existing database system. In a third step, an evaluation is presented regarding issues like recovery and partial replication.

The basic mechanisms behind those protocols are to first perform a transaction locally, deferring writes or performing them on shadow copies. At commit time all updates are sent to all copies in a single message. Those updates are sent using a total order multicast provided by group communication systems (Chapter 4). By

obeying this total order whenever transactions conflict, the global serialization order can be determined individually at each site.

In order to accelerate the execution of the updates at the remote sites, the physical values of the changed data items can be sent, which can simply be applied without re-executing the operation.

2.3 Database Recovery

The goal of database recovery is to restore the failed database to the most recent consistent state, both centrally and in a distributed manner. Database systems invariably undergo different kinds of failures, including system failures and media failures. Database experts refer to system failures when talking about recovery. The solutions are mainly based on hardware redundancy. Therefore, media failures will not be treated in this report.

There are two distinct types of recovery mechanisms, central recovery and distributed recovery as introduced in Chapter 1. In central recovery, a database brings itself to the last consistent state just before failing. Distributed recovery is the process by which the recovering site brings itself to a consistent state with respect to the other sites.

2.3.1 Central recovery

In this report, failure occurs when the processors of the failed server stop working, or when the actual process(s) of the database server crashes. These are generally referred to as system failures. The main memory of the failed site can be unrecoverable with a system failure. Its stable storage, however, remains intact. When the server rejoins the group, parts of the data might be lost, and therefore the database has to undergo central recovery to bring its own copy to a consistent state [7].

Central recovery is usually performed by a dedicated component of the database called the *recovery manager*. The recovery manager is invoked as soon as the

database starts up. The recovery manager of the database guarantees two of the important four ACID properties of databases (Section 2.1), mainly atomicity and durability [7].

The recovery manager has therefore two main responsibilities as illustrated in Figure 2.2.

1. Aborted transactions that are still reflected in the database must be undone (T2 in the figure), and all transactions that were active at the time of failure must be aborted (T3 in the figure). This is the UNDO phase.

2. Committed transactions that were not propagated to stable storage must be redone (T1 in the figure). This is the REDO phase.

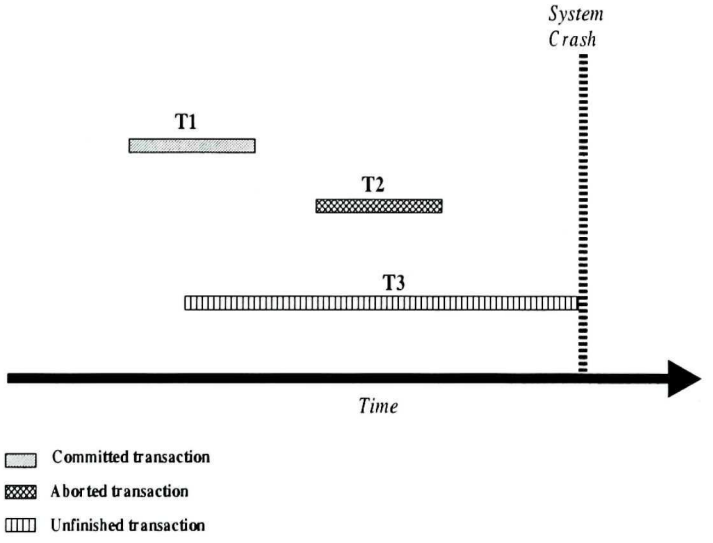


Figure 2.2: Transactions in the UNDO/REDO recovery protocol

Buffering and storing transaction changes

A *page* in the database field is a unit of information storage and retrieval, a *dirty page* refers to a page whose changes are not yet reflected on stable storage. The

tasks of the recovery protocol depend highly on how and when the database system flushes dirty pages to stable storage. There are two considerations here. The first is whether any changes are flushed, i.e. forced to disk, before commit. The second is whether a transaction is allowed to commit before its changes are flushed. This gives the following choices [28]:

1. **No-steal:** pages modified by a transaction must be kept in the buffer up until the end of the transaction. The buffer must be large enough to accommodate dirty pages of even the longest transactions.
2. **Steal:** dirty pages can be written to the stable database before the end of the transaction.
3. **No-force:** a transaction is allowed to commit even if its updates are not written to stable storage. This avoids a sequence of very expensive disk writes.
4. **Force:** a transaction is allowed to commit only after its updates are written to stable storage.

Database systems strive to use the steal, no-force approach because of its efficiency and optimum use of storage resources. The force approach, when compared to the no-force one, has a poor response time. This is due to the fact that the database server has to force to disk every write operation. However, the force approach provides better and a straight forward way to ensure durability. In the same analogy, the buffer manager steals frames from uncommitted transactions to enhance resource usage efficiency. However, the price to pay here is the increasing complexity to ensure atomicity.

A central log is basically a read/append data structure maintained on stable storage to survive failures. When a transaction updates an object, the database stores the old version of the object, or *before-image* in database terminology. When a transaction aborts, the database copies the before-image to the current object location in the database. This is usually referred to as UNDO, i.e. the changes of uncommitted

transactions are rolled back. When a transaction commits, or successfully updates an object, the database stores the new version of the object, or the *after-image*. This new version can be used by the database to REDO the changes of that particular transaction during recovery. Therefore, the database stores two versions of an object in the log, the before-image and the after-image. Additionally, when a transaction starts, a begin record is appended to the log; and when a transaction commits/aborts, a commit/abort record is appended to the log.

Types of recovery

Database systems perform different kinds of recovery, at system startup and during normal processing. There are three scenarios when a database performs recovery of transactions:

- Local UNDO during normal processing: whenever a transaction aborts, the updates of that transaction are undone by installing the before-images.
- Global UNDO at restart after a system crash: transactions that were aborted before the crash should be undone (an abort record will be found in the log), and transactions that were active at the time of the crash should also be undone (In this case, neither an abort nor a commit record will be found in the log). Whenever pages on the disk have updates of the above two transactions, these updates have to be undone by installing the before-images.
- Partial REDO at restart after system crash: transactions that committed before the crash (a commit record will be found in the log). Whenever pages on the disk do not have the updates of such transactions, the updates should be redone by installing the after-images.

2.3.2 Distributed recovery

In a distributed recovery phase, a site has to seek support from another site to get the transactions that it has missed while it was down. Distributed recovery

is discussed in detail in chapter 7. In this section we will introduce the necessary principles to distributed recovery.

Transferring data from one site to another has been discussed in [19]. This paper provides a discussion of online reconfiguration in replicated database systems using group communication. In that paper several protocols for database supported data transfer are analyzed.

Distributed transactions are executed on more than one site. During a transaction, some sites might fail. However, the effects of that transaction are all-or-nothing. This is called *atomic commitment.* The main protocol to ensure atomic commitment of distributed transactions is *Two-Phase Commit* (2PC) protocol [28]. This protocol uses a special site that is called *the coordinator* that coordinates the communication between the sites.

The Two-Phase Commit Protocol

Two-phase commit is an elegant protocol that ensures the atomic commitment by insisting that all sites involved in the execution of a transaction agree to commit the transaction before its effects are made permanent. Figure 2.3 illustrates this protocol. Initially, the coordinator writes a "begin-commit" record in its stable storage, sends a "prepare" message to all participant sites, and enters the WAIT state.

When a participant receives a "prepare" message, it checks if it could commit the transaction. If so, the participant writes a ready record in stable storage, sends a "vote-commit" message to the coordinator and enters the READY state; otherwise the participant writes an abort record and sends a "vote-abort" message to the coordinator, If the decision of the site is to abort it can forget about that transaction, since an abort decision serves as a veto (i.e. unilateral abort). After the coordinator has received a reply from every participant, it decides whether to commit or to abort the transaction. If even one participant has registered a negative vote, the coordinator has to abort the transaction globally. So it writes an abort record,

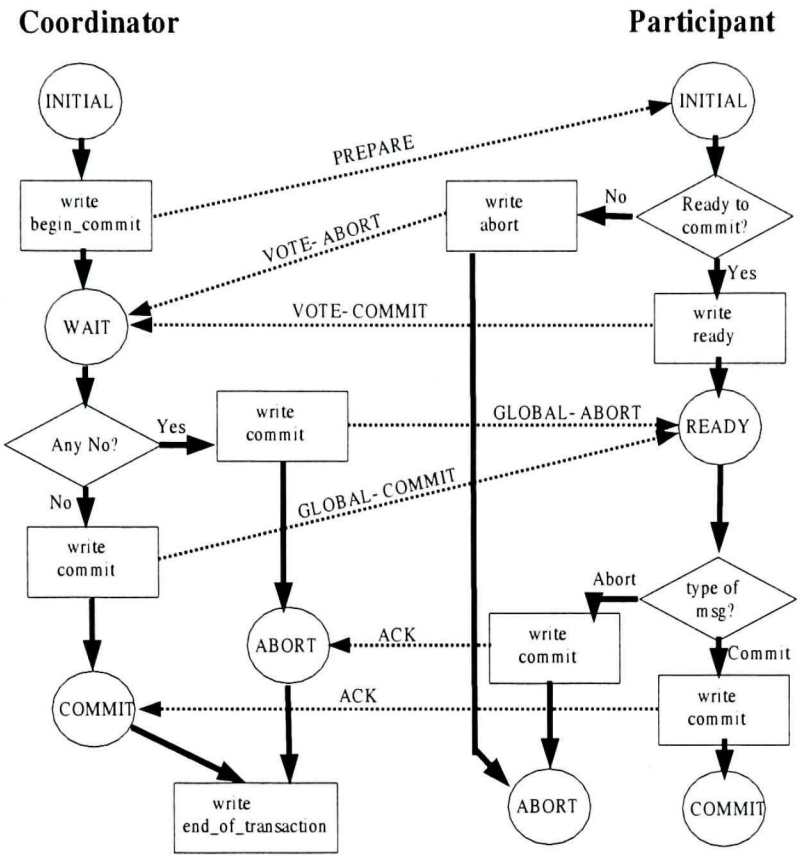


Figure 2.3: The 2PC protocol

sends a "global-abort" message to all participants sites, and enters the ABORT state; otherwise, it writes a commit record, sends a "global-commit" message to all participants, and enters the COMMIT state. The participants either commit or abort the transaction according to the coordinator's instructions and send back an acknowledgement, at which point the coordinator terminates the transaction by writing an end-of-transaction record to stable storage.

Chapter 3

The PostgreSQL Database System

3.1 Introduction

PostgreSQL is an object-relational database management system. The first versions, called Postgres, were developed at the University of California at Berkeley by Professor Michael Stonebraker. PostgreSQL is a descendant of this original Berkeley code. It is now maintained by the open-source community. It is continuously upgraded and new features are added. The current version is PostgreSQL 7.3. It supports the SQL language and is a fully fledged database system. It has wide spread use because of its rich capabilities, including numerous modern database features such as almost all SQL constructs, including subselects and transactions. Furthermore, it has enriched functionality like user-defined types and functions.

3.2 PostgreSQL Architecture

PostgreSQL attributes its success mainly to its modern layer-based and modular architectural design. At the highest level, it has a process-per-user client-server

architecture. The main process, i.e. the one that listens for client connections, is called the *postmaster*, or the database server. When a client, or a *frontend* in PostgreSQL terminology, connects to database server through its listening port, the system forks a *backend* process, and from there on, the communication between the frontend and the database is solely handled by the backend. Figure 3.1 illustrates this top-layer architectural concept.

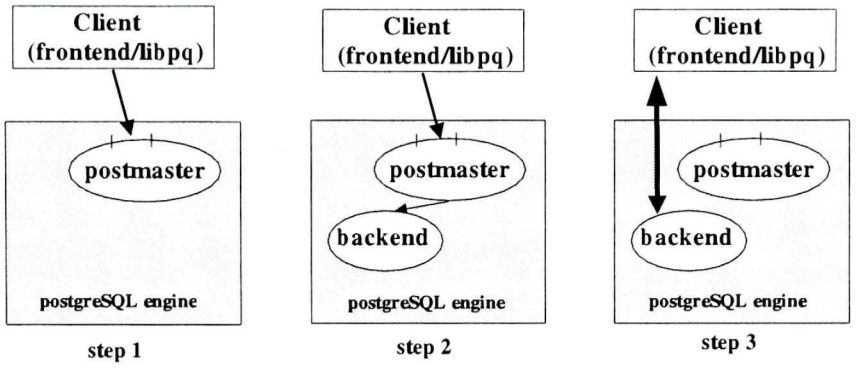


Figure 3.1: PostgreSQL architecture

The communication between client and server is based on a message-based protocol. The protocol is implemented over both TCP/IP and Unix sockets. There is only one PostgreSQL postmaster running per system, however there are as many backends as there are connected clients.

The underlying engine of PostgreSQL, however, is quite complex. It is based on various distinct modules [10], to maintain a desirable modular aspect. This is illustrated graphically in Figure 3.2.

At startup, PostgreSQL initializes the database by creating a template database, the cache, the transaction log and various other system initializations. The postmaster is basically a daemon that is launched at startup. The postmaster handles all user connections, manages system wide operations such as startup, shutdown, periodic checkpointing, statistics collecting, etc ... It resets the system if a backend crashes and cleans after the crashed process. It also creates the shared memory and

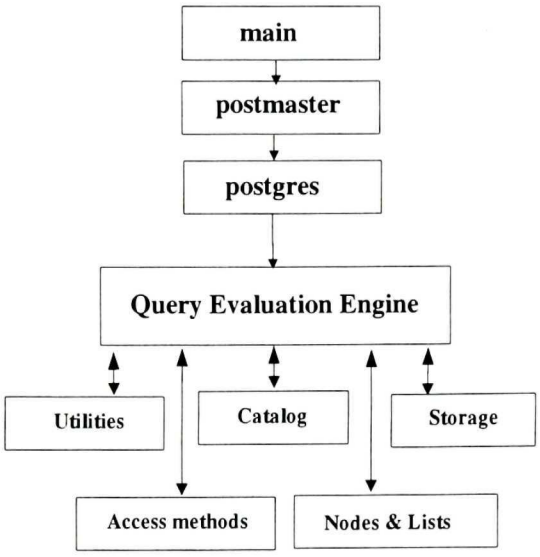


Figure 3.2: PostgreSQL architecture

semaphore pools during startup for other subsystems to use.

The File and Access Methods subsystem provides support for various data access methods, including hashing, heap for data rows, B-Tree for Lehman and Yao's btree management algorithm [23], R-Tree for indexing of 2-dimensional data and index.

The Storage Management Subsystem manages various storage systems. They include the shared buffer pool manager, the file manager, the semaphores and shared memory management, the large objects handler, the lock manager, the page manager and the storage disk manager. The only supported storage manager is the magnetic disk manager. PostgreSQL uses the term *resource manager* as a global name to refer to these managers. Every relation in the system is tagged with the storage manager on which it resides. The storage manager switch code turns what used to be filesystem operations into operations on the correct store, for any given relation.

The Library subsystem provides data structures and functions shared by other subsystems. Major components include utilities, regular expression and other support functions. The utilities module includes support for built in data types and various

relation caches.

The Query Evaluation Engine contains the PostgreSQL backend main handler, as well as the code that makes calls to the parser, optimizer, executor, and commands functions.

3.3 PostgreSQL Transaction Processing

There are two types of transactions in Postgres-RFR, *single statement transactions* and *multiple statement transactions.* A single statement transaction is not enclosed between the conventional 'BEGIN' and 'END'/'COMMIT' clauses. Single statement transactions are standalone, single line statements such as 'CREATE TABLE T ...'. PostgreSQL automatically creates a transaction before the execution of the statement, and terminates the transaction after the execution. Multiple statement transactions, on the other hand, can contain more than one SQL statement. They have to be enclosed by the above clauses.

When a user connects to the database through a client, PostgreSQL launches a local backend to handle the connection. The local backend is a process that listens for commands from the user and acts upon them. Depending on the request, the local backend starts a transaction and interacts with the database to fulfill the user's request. This is represented schematically in Figure 3.3. The local backend acts on every command issued by the user. If a transaction is not already started at the reception of a command, the local backend starts one, by creating a new transaction number, updating the lock table, the cache and memory control structures. For every query, the local backend performs basic parsing and creates a parse tree list. It then processes every one of these parse trees by performing parsing analysis and rule rewriting, and creates a query tree list. It then runs through these queries and processes them one at a time.

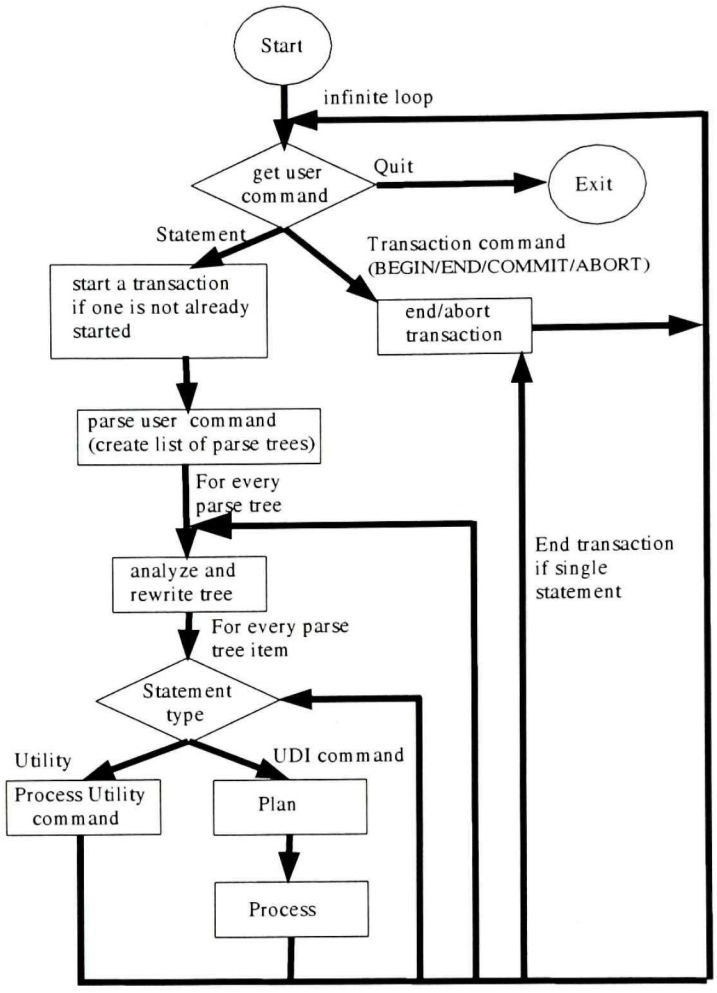


Figure 3.3: Transaction processing loop

3.4 PostgreSQL MVCC Concurrency Control Protocol

PostgreSQL uses a multi-version concurrency control, or MVCC, to handle concurrency. In PostgreSQL, MVCC entails that each transaction sees a snapshot of the data (a database version) when it starts, regardless of what the other concurrent transactions are doing. This enforces the isolation criteria of the ACID properties (Section 2.1). With this protocol, readers and writers never block each other during

execution. It is only at commit time that a concurrency mechanism is run to serialize the updates to the same row. The sensitive issue here is that if two transactions start out with the same tuple version and both update it, then a protocol is required to store the combined effect of both transactions, with the second transaction properly invalidating the update of the first logically concurrent transaction.

PostgreSQL offers two protocols [10] to the issue of concurrent updates to the same row:

- Read committed level: allow the second transaction to use the new tuple as input values. This requires changing tuple visibility.
- Serializable level: abort the second transaction all together. The client application must redo the whole transaction, which will then be allowed to see the new value of the tuple under strict serializable-behavior rules.

The above protocol requires storing multiple versions of every row. PostgreSQL uses non overwriting storage management, which means that updated tuples are appended to the table, and older versions are removed sometime later. Currently, removal of long-dead tuples is handled by a special 'vacuum' maintenance command that must be issued periodically.

3.5 PostgreSQL Write Ahead Log

Write Ahead Logging (WAL) is a standard approach to transaction logging. The core concept is that changes to the database must be written only after those changes have been logged, i.e. when log records have been flushed to permanent storage. With WAL, data pages don't need to be flushed to disk on every transaction commit, because in the event of a crash, the data could be recovered using the log. Any changes that have not been applied to the data pages will first be redone from the log records and then changes made by uncommitted transactions will be removed

from the data pages. Furthermore, the log has to be written before the transaction commits.

3.5.1 Benefits of the WAL

WAL allows for a significant reduction in the number of disk writes, since only the log file needs to be flushed to disk at the time of transaction commit. Furthermore, the log file is written sequentially, and so the cost of updating the log is much less than the cost of flushing the data pages.

3.5.2 Implementation of the WAL

WAL logs are stored in a special directory, the *data directory*, where all the WAL information as well as the user data is stored. The logs are stored as a set of segment files, each 16 MBytes in size. Each segment is divided into 8 KBytes pages. Segment files are given ever-increasing numbers as names, starting at 0000000000000000. The numbers do not wrap, at present, but it should take a very long time to exhaust the available stock of numbers.

The WAL buffers and control structure are in shared memory, and are handled by the backends. They are protected by lightweight locks, which are locks used to manage access to data structures in shared memory. The demand on shared memory is dependent on the number of buffers. The default size of the WAL buffers is 8 buffers of 8 KBytes each, or 64 KBytes total.

3.5.3 PostgreSQL recovery with WAL

Central recovery with PostgreSQL uses the concept of checkpointing. Checkpoints are points in the sequence of transactions at which it is guaranteed that the data files have been updated with all information logged before the checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. As a result, in the event of a crash, the recovery manager

knows from what record in the log it should start the REDO operation, since any changes made to data files before that record are already on disk. After a checkpoint has been made, any log segments written before the undo records are no longer needed and can be recycled or removed.

The postmaster spawns a special backend process every so often to create the next checkpoint. A checkpoint is created once a predefined number of log segments have been used, or once a predefined lapse of time, whichever comes first. The default settings are 3 segments and 300 seconds respectively. It is also possible to force a checkpoint by using the SQL command CHECKPOINT.

When recovery is to be done, the backend first reads the checkpoint record; then it performs the REDO operation by scanning forward from the log position indicated in the checkpoint record. Because the entire content of data pages is saved in the log on the first page modification after a checkpoint, all pages changed since the checkpoint will be restored to a consistent state.

3.5.4 The WAL record format

A record in the WAL is stored in two parts, the record header and the body. The log record header contains seven fields as illustrated in Figure 3.4.

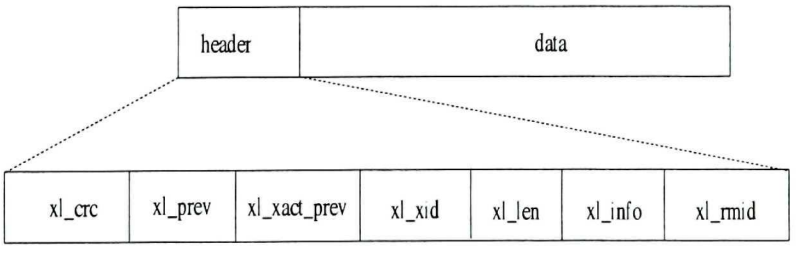


Figure 3.4: PostgreSQL WAL record structure

where:

- **xl_crc:** a CRC for this record (PostgreSQL internal functionality).

- **xl_prev:** a pointer to a previous record in the log
- **xl_xact_prev:** a pointer to the previous record of the same transaction
- **xl_xid:** the transaction ID
- **xl_len:** total length of the resource manager data
- **xl_info:** flag bits, or control bits
- **xl_rmid:** the resource manager for this record

This is illustrated graphically in Figure 3.5.

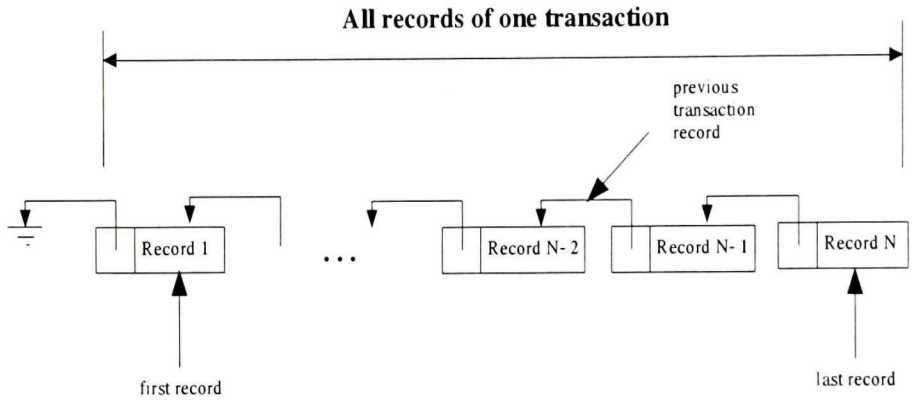


Figure 3.5: Transaction central logging

3.5.5 The WAL interface

WAL is a logging mechanism that provides an interface to interact with it. This interface is composed of variables and functions. At the heart of the WAL mechanism is a control structure called the *Control File*. The control file holds important information pertinent to WAL such as pointers to the last two checkpoint records, and the database state at the time of shutdown, i.e. was the database properly shutdown or was it as a consequence of a crash. A second checkpoint is required in the case when the last one is unreadable or corrupt. The following is a summary of the important functions the WAL provides:

- **XLogInsert:** This function takes a record and stores it in the log. This function returns a pointer to the end of the inserted record.
- **ReadRecord:** This attempts to read a log record, at the specified location
- **StartupXLOG:** This must be called only once, and only at database startup. When called, it reads the control file, and carries out the REDO/UNDO central recovery mechanism.
- **ShutdownXLOG:** This must be called only once, and only at database shutdown. When called, it clears the temporary log structures, such as the log caches, and inserts a checkpoint in the log.
- **CreateCheckPoint:** This creates and forces a checkpoint record in the log.
- **ReadCheckpointRecord:** This reads a checkpoint from the log. WAL gives the option to read up to the last two checkpoint records. This is useful if the last checkpoint is corrupt or not valid.
- **GetRedoRecPtr:** This gets the record from which the REDO has to be performed.
- **GetUndoRecPtr:** This gets the record from which the UNDO record has to be performed.
- **ReadControlFile:** This reads the control file
- **UpdateControlFile:** This forces a copy of the control file to be written to disk.

3.6 PostgreSQL commands

PostgreSQL has a wealth of commands to handle the user's request. There are three types of commands in PostgreSQL, *IDU*, *select* and *Utility* commands. An IDU command is one of three SQL commands that involve a write instruction to

the user's data. These include Insert, Delete and Update. We will pay special attention to PostgreSQL commands in the replication chapter (Chapter 5) of this report, so a detailed description of all commands is due in this section.

3.6.1 The IDU commands

There are three IDU commands, namely insert, delete and update.

	Category	Command	Description
1	IDU commands	INSERT	Create a new row(s) in a table
2		DELETE	Delete row(s) of a table
3		UPDATE	Update row(s) of a table

3.6.2 The select command

The 'SELECT' command will return rows from one or more tables.

	Category	Command	Description
1	Select Command	SELECT	Retrieve rows from a table or view

3.6.3 The utility commands

There are 65 Utility commands in total. These commands provide transaction, user, database, groups, tables, and views management, as well as user defined types, triggers, functions, languages and sequences commands.

Transaction management commands

Users can set the behavior of constraint evaluation in the current transaction using the 'SET CONSTRAINTS' command. They can choose to have the constraints checked at the end of each statement or until transaction commit. The 'SET

TRANSACTION' command sets the transaction isolation level. The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently. Transaction management commands are summarized in the following table:

	Category	Command	Description
1	Transaction Management Commands	BEGIN	Start a transaction block
2		COMMIT	Commit the current transaction
3		ABORT	Abort the current transaction
4		ROLLBACK	Abort the current transaction
5		END	Commit the current transaction
6		SET CONSTRAINTS	Set the constraint mode of the current transaction .
7		SET TRANSACTION	Set the characteristics of the current transaction

User data management commands

PostgreSQL has a rich set of user's data management commands. These include database, table, view and index commands. The special 'VACUUM' command reclaims storage occupied by deleted tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by UPDATE are not physically removed from their table (updates create new versions of tuples instead of overwriting the old ones); they remain present until a VACUUM is done. Therefore it's necessary to do VACUUM periodically, especially on frequently-updated tables. The 'LOCK' command allows a user to control concurrent access to a table for the duration of a transaction. PostgreSQL always uses the least restrictive lock mode whenever possible. The 'LOCK' command provides for cases when the user might need more restrictive locking. A summary of these commands is provided below:

	Category	Command	Description
8	Database Commands	CREATE DATABASE	Create a new database
9		DROP DATABASE	Remove a database
10		VACUUM	Garbage-collect and optionally analyze a database
11	Table Commands	CREATE TABLE	define a new table
12		CREATE TABLE AS	Create a new table from the results of a query
13		DROP TABLE	Remove a table
14		ALTER TABLE	Change the definition of a table
15		TRUNCATE	Empty a table
16		SELECT INTO	Create a new table from the results of a query
17		COPY	Copy data between files and tables
18		LOCK	Explicitly lock a table
19		CLUSTER	Cluster a table according to an index
20	View Commands	CREATE VIEW	Define a new view
21		DROP VIEW	Remove a view
22	Index Commands	CREATE INDEX	Define a new index
23		DROP INDEX	Remove an index
24		REINDEX	Rebuild corrupted indexes

User, group and access management commands

Only super-users are allowed to use the group and user management commands. The 'GRANT' command gives specific permissions on an object (table, view, sequence) to one or more users or groups of users. These permissions are added to those already granted, if any. Users other than the creator of an object do not have any access privileges to the object unless the creator grants permissions. There is no

need to grant privileges to the creator of an object, as the creator automatically holds all privileges. The 'REVOKE' command allows the creator of an object to revoke previously granted permissions from one or more users or groups of users. Note that any particular user will have the sum of privileges granted directly to him, privileges granted to any group he is presently a member of, and privileges granted to a special group called 'PUBLIC'. These commands are summarized below:

	Category	Command	Description
25	Group Commands	CREATE GROUP	Define a new user group
26		DROP GROUP	Remove a user group
27		ALTER GROUP	Add users to a group or remove users from a group
28	User Commands	CREATE USER	Define a new database user account
29		DROP USER	Remove a database user account
30		ALTER USER	Change a database user account
31	Access Commands	GRANT	Define access privileges
32		REVOKE	Remove access privileges

User-defined features commands

PostgreSQL allows function overloading; that is, the same name can be used for several different functions as long as they have distinct argument types. This facility must be used with caution for internal and C-language functions, however. PostgreSQL allows users to register a new procedural language with a PostgreSQL database. Subsequently, functions and trigger procedures can be defined in this new language. The user must have the PostgreSQL super-user privilege to register a new language. Creating a language effectively associates the language name with a call handler that is responsible for executing functions written in the language. Creating a trigger can be specified to fire either before the operation is attempted on a tuple or after the operation has been attempted. If the trigger fires before the

event, the trigger may skip the operation for the current tuple, or change the tuple being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are "visible" to the trigger. Creating a TYPE allows the user to register a new user data type with PostgreSQL for use in the current database. This requires the registration of two functions (using CREATE FUNCTION) before defining the type. Creating an AGGREGATE allows a user or programmer to extend PostgreSQL functionality by defining new aggregate functions. Some aggregate functions for base types such as min(integer) and avg(double precision) are already provided with PostgreSQL. If one defines new types or needs an aggregate function not already provided, then this command can be used to provide the desired features. Creating an OPERATOR allows the user to define a new operator. The PostgreSQL rule system allows one to define an alternate action to be performed on inserts, updates, or deletions from database tables. Rules are used to implement table views as well. The semantics of a rule is that at the time an individual instance (row) is accessed, inserted, updated, or deleted, there is an old instance (for selects, updates and deletes) and a new instance (for inserts and updates). All the rules for the given event type and the given target object (table) are examined, in an unspecified order. Creating a SEQUENCE will enter a new sequence number generator into the current database. This involves creating and initializing a new single-row table with the name *seqname*. The generator will be owned by the user issuing the command. After a sequence is created, the user has to use the functions *nextval*, *currval* and *setval* to operate on the sequence.

These commands are summarized below:

	Category	Command	Description
33	Function	CREATE FUNCTION	Define a new function
34	Commands	DROP FUNCTION	Remove a user-defined function
35	Language	CREATE LANGUAGE	Define a new procedural language
36	Commands	DROP LANGUAGE	Remove a user-defined procedural language
37	Trigger	CREATE TRIGGER	Define a new trigger
38	Commands	DROP TRIGGER	Remove a trigger
39	Type	CREATE TYPE	Define a new data type
40	Commands	DROP TYPE	Remove a user-defined data type
41	Aggregate	CREATE AGGREGATE	Define a new aggregate function
42	Commands	DROP AGGREGATE	Remove a user-defined aggregate function
43	Operator	CREATE OPERATOR	Define a new operator
44	Commands	DROP OPERATOR	Remove a user-defined operator
45	Rule	CREATE RULE	Define a new rewrite rule
46	Commands	DROP RULE	Remove a rewrite rule
47	Sequence	CREATE SEQUENCE	Define a new sequence generator
48	Commands	DROP SEQUENCE	Remove a sequence

Curser management commands

Users can create cursors using the 'DECLARE' command, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary format using the 'FETCH' command. The 'MOVE' command allows a user to move a cursor position a specified number of rows. The 'CLOSE' command frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. These commands are summarized below:

	Category	Command	Description
49	Curser Commands	DECLARE	Define a cursor
50		FETCH	Retrieve rows from a table using a cursor
51		MOVE	Position a cursor on a specified row of a table
52		CLOSE	Close a cursor

Run time configuration commands

PostgreSQL allows users to change run-time configuration parameters, such as encoding, date style and time zones. The 'RESET' command restores run-time parameters to their default values, and the 'SHOW' command will display the current setting of a run-time parameter. This is summarized below:

	Category	Command	Description
53	Run-time Commands	SET	Change a run-time parameter
54		RESET	Restore the value of a run-time parameter to a default value
55		SHOW	Show the value of a run-time parameter

Notification commands

The 'LISTEN' command registers the current PostgreSQL backend as a listener on any notify condition. Whenever the command 'NOTIFY' is invoked, either by this backend or another one connected to the same database, all the backends currently listening on that notify condition are notified, and each will in turn notify its connected frontend application. A backend can be unregistered for a given notify condition with the 'UNLISTEN' command. Also, a backend's 'LISTEN' registrations are automatically cleared when the backend process exits. This is summarized below:

	Category	Command	Description
56	Notifications Commands	LISTEN	Listen for a notification
57		UNLISTEN	Stop listening for a notification
58		NOTIFY	Generate a notification

Session management commands

The following commands are used to display session parameters.

	Category	Command	Description
59	Session Commands	CURRENT_DATE	
60		CURRENT_TIME	
61		CURRENT_TIMESTAMP	
62		CURRENT_USER	

Miscellaneous commands

The 'COMMENT' command stores a comment about a database object. The 'EXPLAIN' command displays the execution plan that the PostgreSQL planner generates for the supplied query. The execution plan shows how the table(s) referenced by the query will be scanned, either by plain sequential scan, index scan, etc. and if multiple tables are referenced, what join algorithms will be used to bring together the required tuples from each input table. The most critical part of the display is the estimated query execution cost, which is the planner's guess at how long it will take to run the query. The 'LOAD' command loads a shared library file into the PostgreSQL backend's address space. If the file had been loaded previously, it is first unloaded. This command is primarily useful to unload and reload a shared library file that has been changed since the backend first loaded it. To make use

of the shared library, function(s) in it need to be declared using the 'CREATE FUNCTION' command. These commands are summarized below:

	Category	Command	Description
63	Miscellaneous Commands	COMMENT	Define or change the comment of an object
64		EXPLAIN	Show the execution plan of a statement
65		LOAD	Load or reload a shared library file

Chapter 4

The Spread Group Communication System

4.1 Introduction to Group Communication Systems

Group communication systems, or *GCS*, are application-level multicast techniques that provide a set of communication services. These services include *message guarantees* in transmitting messages from one application to another across a network, as well as *group membership services*. GCS provide two message guarantees, namely *reliable* and *ordered* message delivery. Reliable means that messages are delivered despite failures, and ordered delivery guarantees that messages are received in special order, such as FIFO order.

The main drive for group communication research at the early stages were high availability and fault tolerance [31]. A fault-tolerant system is designed to keep running even after a fault has occurred. Fault-tolerant systems use a variety of tools to ensure high availability, including redundancy and replication mechanisms. A number of early systems were developed at this early stage, such as ISIS [8], Horus [31], Transis [11], Totem [25], RMP [33] and Newtop [12]. These systems were initially designed for local area networks and therefore had scalability and

security setbacks [3]. Recent work in this area focused on scaling and securing group membership, in order to facilitate their deployment to wide area networks.

A *group* is a logical set of sites, or computers, that exchange messages between them to perform distributed tasks. A site is said to be a *member* of a group if it is up and running, and is able to send and receive messages within that group. A *view* is a snapshot of the memberships of all sites in the system at a particular moment. A *membership change*, or view change, is a special message sent by the GCS to the members that are up and running to advise them that some site(s) has joined or left the group. When a site sends a message to the group, the GCS ensures that it is received by all members of that same group, including the sender, a technique referred to as *multicast*.

4.2 Virtual Synchrony

Virtual Synchrony (VS) is a concept that was derived from the Virtual Synchrony Model defined by Ken Birman, et. al. in their early work on the ISIS system [8], one of the early group communication systems. Virtual synchrony is able to handle message omission failures, as well as fail-stop process failures. Virtual Synchrony guarantees that membership changes within a group are observed in the same order by all the group members that remain connected. Moreover, membership changes, or view changes, are *totally ordered* (Section 4.4.2) with respect to all other regular messages that the system has to handle. This means that every two members that observe the same two consecutive membership changes, receive the same set of regular multicast messages between the two changes.

Virtual synchrony states that:

1. There is a unique view in a properly functioning system, on which all members of that group agree.

2. If a message m is multicast in view v before view change vc, then either no

member in v that executes vc ever receives m, or every member in v that executes vc receives m before performing vc.

This definition imposes a *total order* (Section 4.4.2) between view changes and multicast messages, but does not enforce any ordering between messages delivered in the same view.

4.3 Extended Virtual Synchrony

A restriction to virtual synchrony is that it is not able to handle network partitions. Due to the asynchronous nature of the system model, a safe conclusion as to which of the messages were received by which members just before a network partition occurs is impossible to confirm with VS. In fact, it has been proven that reaching consensus in an asynchronous environment, with the possibility of even one failure is impossible [13]. Hence group communication primitives based on Virtual Synchrony do not provide any guarantees of message delivery that span network partitions.

Extended Virtual Synchrony (EVS) [24] is an enhanced Virtual Synchrony model that tries to solve the above problem with network partition. Extended Virtual Synchrony divides the view change message into two parts, a transitional view change message and a regular view change message. The transitional view change message contains the members that move from the old regular view to the next regular view. This protocol dictates that the GCS does not immediately deliver the view change to the application, but rather switches into a transitional phase, and tries to recover any lost messages, i.e. messages that were not delivered to all view members, from the previous view. This way, uniformity and consistency among members that are still connected is achieved. Meanwhile it does not multicast new messages from the application, but buffers them until the transitional phase ends and the new view is established. The new view starts as soon as the GCS delivers the new membership change to the application. Only then that the GCS multicasts the buffered messages.

4.4 Spread

Spread is a general-purpose GCS for local and wide area networks [2]. It provides reliable and ordered delivery of messages, as well as a membership service. Reliability is a general term used by the research community that encompasses FIFO, causal and total ordering of messages as detailed below. The system consists of a server and a client library linked with the application.

Spread offers a secure many-to-many communication paradigm [3], where any group member can be both a sender and a receiver of messages. Although designed to support small to medium size groups, it can accommodate a large number of group sizes, each spanning the Internet. Spread scales well with the number of groups used by the application without imposing any overhead on network routers [2].

4.4.1 Spread architecture

The Spread system is based on a daemon-client model where clients connect to one of the system daemons to gain access to the group communication services. The daemons constitute the backbone, or system managers, which are responsible for establishing the message network infrastructure, and provide basic membership and ordering services. Clients usually connect using a small client library, and can reside anywhere on the network [2].

Spread is highly configurable, allowing the user to customize the system according to their needs. Spread can be configured to use just one daemon or to use one daemon in every machine running group communication applications. The best performance, in the absence of failures, is achieved when a daemon is on every machine.

All daemons participating in a Spread configuration know the complete *potential* memberships of all members when started. The knowledge of the actual memberships of active daemons is gathered dynamically during operation.

Spread supports the Extended Virtual Synchrony model [24] of group membership.

As explained above, EVS can handle network partitions and re-merges, as well as joins and leaves.

4.4.2 Spread Reliable Multicast Services

Spread provides three message delivery semantics, *unreliable*, *reliable* and *uniform reliable*. It also provides four message ordering semantics, *unordered*, *FIFO*, *causal* and *total* order [2].

The reliability semantics are:

- **Unreliable:** Unreliable message delivery is the simplest form of message exchange. There is no guarantee that the message will be delivered to the intended recipient. The sender transmits the message and if the target does not receive the message, the sender will not re-transmit the message.
- **Reliable:** In contrast to the unreliable message guarantee, reliable messages are guaranteed to be delivered to the application with Spread. Reliable messages provide the following delivery guarantee: whenever a message is delivered to a site and this site does not fail for sufficiently long time, then all other sites will deliver the message unless they fail. Reliable messages use the network layer, to achieve reliability. Each link in the network guarantees reliable transport within a bounded time, in the absence of processor or network faults. Thus end-to-end reliability is provided in the case where there are no faults, because eventually every packet will make it across all the links to all the daemons which need it.
- **Uniform reliable:** Uniform reliable delivery guarantees that whenever a site N delivers a message to the application , all other sites will deliver the message unless they fail (even if N fails immediately after the delivery). This requires that a message is not delivered before all sites have acknowledged that they have physically received the message. Hence it causes a higher delay than reliable delivery.

The message ordering semantics are:

- **Unordered:** Unordered messages are delivered as soon as the complete message is received since they do not provide any ordering guarantees. Therefore, their delivery is never delayed due to other messages.
- **FIFO:** FIFO messages provide the same reliability guarantee as reliable messages. Spread assigns sequence values to messages to ensure FIFO ordering. Spread guarantees that a FIFO message will be delivered only after all messages with lower sequence values have been delivered. Additionally, the messages of each sender are delivered in the order they are sent.
- **Causal:** This delivery semantic guarantees that the message order preserves the happens-before relation (after Lamport [22]). It includes FIFO order, and guarantees that a response to a message is never delivered before the message that caused it.
- **Total:** With total order, all sites receive all messages in the same total order, independent of who sent them. Additionally, total order is consistent with both FIFO and causal ordering, i.e. messages that contradict the causality or the FIFO principles are not delivered.

Figure 4.1 illustrates the relative cost of the various services.

Not all message guarantees can be combined with all message orderings. In fact, the only possible combinations are: unreliable/not-ordered, reliable/not-ordered, reliable/FIFO, reliable/causal, reliable/total (called agreed in Spread terminology) and uniform-reliable/total (called safe with Spread).

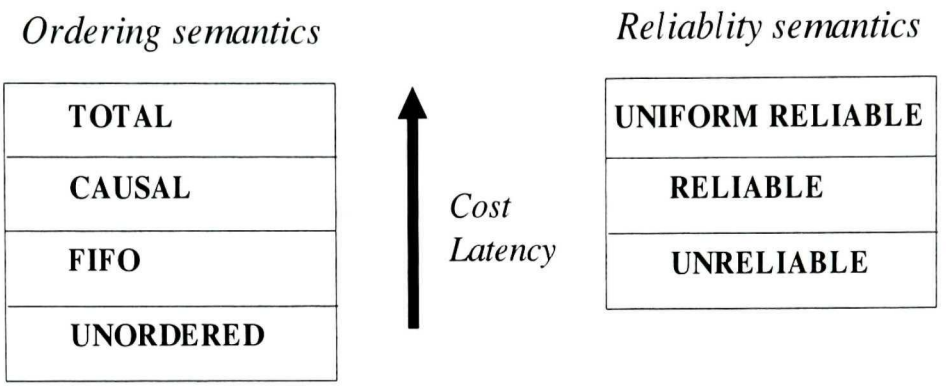


Figure 4.1: Reliable Multicast Services

Chapter 5

Synchronous Master-Slave Replication

5.1 Introduction

The replication mechanism used in this work is a master-slave eager replication where each site has a full copy of the database. Chapter 2 has a detailed explanation of these terms. The design of the replication module is adopted largely from work by Bettina Kemme [21] and Win Bausch [6] performed on Postgres-R, the modified system of PostgreSQL v6.4 with full replication functionality. We will call the new system Postgres-RFR because it includes **R**eplication, **F**ailover and distributed **R**ecovery functionality. The original Postgres-R was based on PostgreSQL v6.4, but the current version of PostgreSQL is 7.2 with a lot of changes, in particular concurrency control and logging. Postgres-R is an update-everywhere synchronous system, and many of the design features of that system were kept and preserved in the new system.

5.2 Replication in Postgres-RFR

The architecture of Postgres-RFR is based on Postgres-R. It includes three dedicated processes that are called the *replication manager*, the *communication manager* and the *remote backend*. These processes are created at system startup, and run as long as the database is up and running. The general architecture is outlined in Figure 5.1. As described below, the main function of the replication manager is to manage the replication cycle. The communication manager handles all message exchanges between the database and the group communication system. The remote backend at the slaves is responsible for applying the updates, sent by the master, at the slaves. The remote backend will have further functionality at the master during distributed recovery as described in Chapter 7.

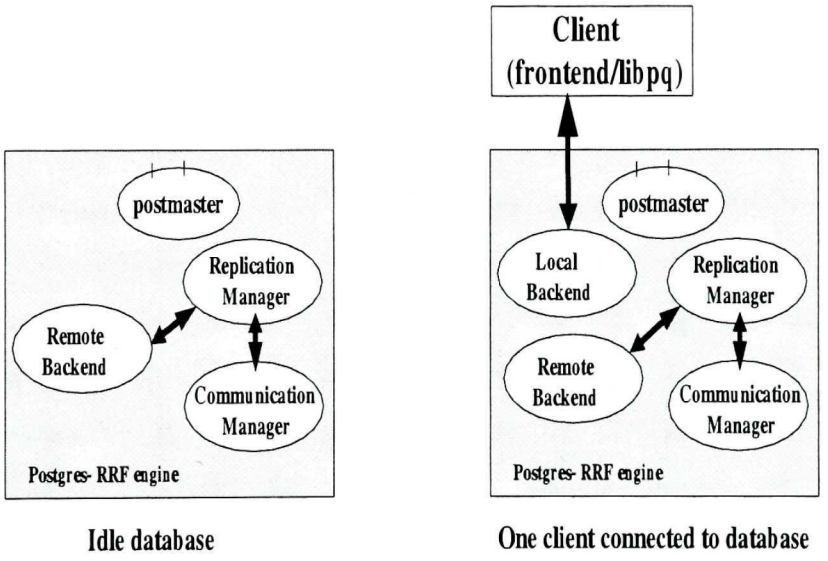


Figure 5.1: Postgres-RFR architecture

Master-Slave replication means that IDU commands (section 3.6) are only allowed at the master site. Queries, i.e. transactions that only display and don't change data, are allowed everywhere, i.e. on all sites including the master. Users have to know which of the sites in the replication group is the master, and have to connect to it directly to apply changes to the database. The master first executes the transaction.

At the end of the transaction, it sends all changes to the slaves which apply them to their copies.

5.2.1 The replication cycle

Replication in Postgres-RFR follows a cycle that starts and ends at the local backend. As before, i.e. with PostgreSQL (Chapter 3), when a client application connects to the database server, this latter forks a local backend to handle the interaction between the user and the database. The transaction is identical to the transaction processing with PostgreSQL if it only contains queries. Replication comes in the picture when there are IDU commands or Utility commands (section 3.6). Figure 5.2 illustrates the different steps in the replication cycle.

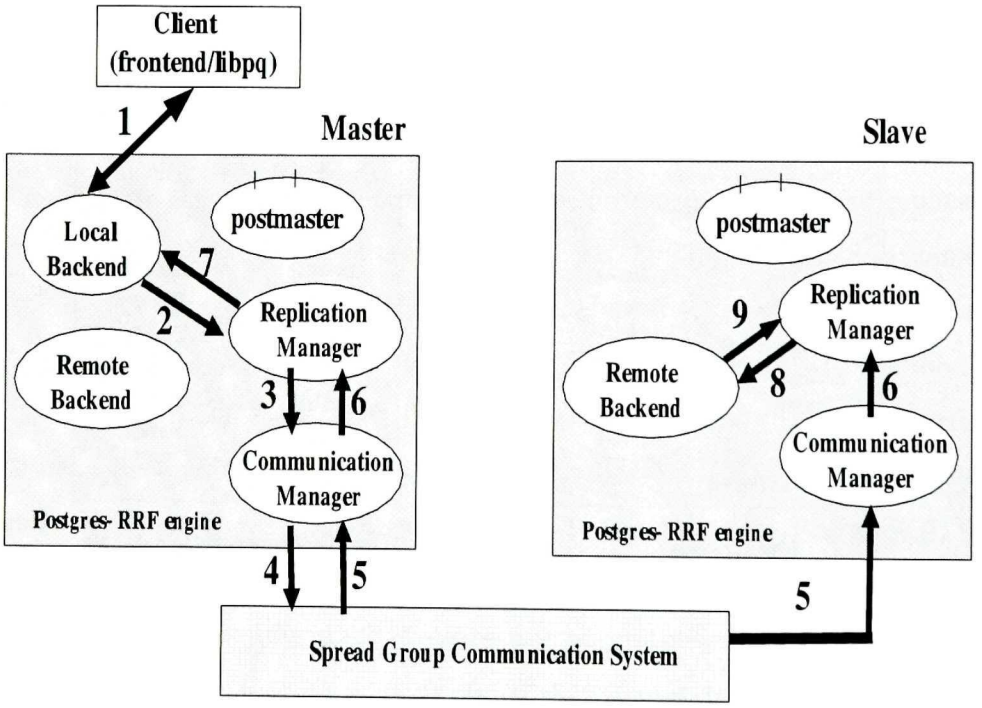


Figure 5.2: Replication Cycle in Postgres-RFR

5.2.2 Replication at the master

Replication starts at the start of a transaction (Section 2.1). The writeset is sent at the end of the successful ending of the transaction, i.e. the transaction does not get aborted. This happens in two scenarios:

- When the user issues an explicit 'COMMIT' command or a 'END' command (Section 2.1) at the end of a multiple statement transaction. Within such a transaction, a 'ROLLBACK' or 'ABORT' will clear all replication temporary structures, and the cycle is never started.
- With a single statement transaction.

As illustrated in Figure 5.2, the steps of replication at the master site are the following:

1. The user's client application interacts with the local backend through transactions. This is step 1 in the figure. In the figure, IDU statements and some utility statements lead to the creation of a writeset (Section 5.3.2). While executing the transaction, the local backend collects the writeset to be sent to the slaves. After the execution of each IDU command, it collects the affected tuples and adds them to the writeset. Alternatively, it simply adds the SQL statement to the writeset (for details, see Section 5.3.2).
2. The local backend opens a communication channel with the replication manager and sends the writeset. This communication channel remains established until the user quits the client at the application level. In this case, it is torn down. The same communication channel is used for any subsequent update transaction from the client. This is step 2 in the figure.
3. When the replication manager receives the update transaction, in the form of a writeset (section 5.3.2), it multiplexes it to the GCS (Chapter 4). This is step 3 in the figure.

4. The GCS broadcasts the writeset to the Spread system in total order. This is step 4 in the figure.

5. The GCS eventually receives the writeset back from the Spread system. This is step 5 in the figure.

6. When the group communication manager receives back the writeset from the Spread system, it simply forwards it to the replication manager. This is step 6 in the figure.

7. The replication manager simply advises the local backend by sending it a special message (Section 5.3.3). The local backend needs this special message to commit the transaction, in agreement with the synchronous nature of the replication protocol. This is step 7 in the figure.

At the end of this cycle, the update transaction is committed at the master. Note that the master site does not need a special confirmation message from the slaves. Instead, by receiving its own writeset back (in step 6 above) the master can safely assume that all slaves will receive (and then apply) the writeset unless they fail. This delivery guarantee is provided by the safe multicast of the Spread group communication system (see Chapter 4).

5.2.3 Replication at the slaves

At the slaves, the cycle of replication is quite different, as depicted in Figure 5.2. Transactions received from the master are processed in FIFO order. There is only one remote backend to handle the processing of the writesets at the slaves. The reason the backend executes writesets serially is simplicity and to guarantee serializability. When the master sends a writeset of transaction T1 before the writeset of transaction T2, then the MVCC concurrency control (Chapter 2) guarantees that T1 is serialized before T2 at the master. T1 will arrive and has to be serialized before T2 at the slave. Theoretically, if they don't conflict, the slave could execute

them concurrently or even T2 before T1 and we would still have serializability. Only if they conflict, the slave has to execute T1 before T2. But this would require to check whether the two transactions conflict. Hence, by executing T1 serially before T2, we ensure that we always have serializability. Local clients that connect to a slave are allowed some operations, such as read operations. Section 5.4 lists in detail the operations that are allowed on the slaves.

As illustrated in Figure 5.2, the steps of replication at the slave sites are the following:

1 The Spread system delivers the writeset to the group communication manager of the slave database server. This is step 5 in the figure.

2 The group communication manager forwards the writeset to the replication manager. This is step 6 in the figure.

3 The replication manager transfers the writeset to the only remote backend for processing. This is step 8 in the figure.

4 The remote backend processes the writeset, and when finished, sends a 'ready' (section 5.3.3) to the replication manager. This is step 9 in the figure.

5.3 Implementation Details

5.3.1 Replication State Machine

Replication is governed by a state machine, that allows the replication manager complete control over the replication mechanism. The replication manager plays a central role in the replication mechanism. It is the process that acts as the multiplexor of writesets and other replication messages between the backends and the group communication process. On the master, the state machine as seen by the replication manager is illustrated in Figure 5.3.

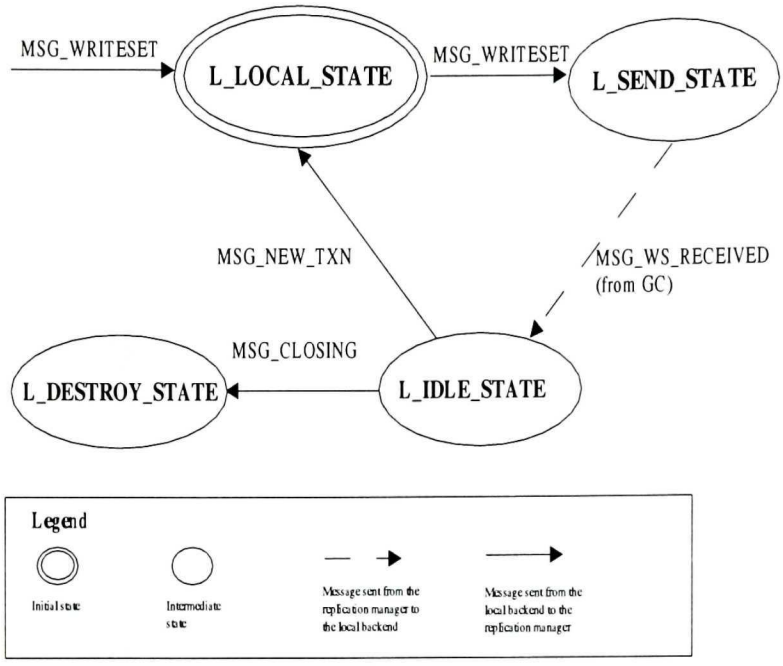


Figure 5.3: Replication state machine at the master

At the master site, the replication manager is unaware of any local backend until they request to connect to it. The first message sent by the local backend is a 'MSG_OPENING'. After this, the replication manager changes the state of the local backend to 'L_LOCAL_STATE'. The only reason why a local backend would request a connection from the replication manager is to send a writeset to the slaves. So, the local backend follows its connection request with the actual writeset ('MSG_WRITESET' in the figure). Upon receiving the writeset, the state of that local backend becomes 'L_SEND_STATE'. At that stage, the replication manager sends the writeset to the group communication manager. The latter broadcasts the writeset to all sites. Upon receiving the writeset back, the replication manager at the master site permits the local backend to commit the transaction by sending it a special message, 'MSG_WS_RECEIVED' and changes the state of the backend to 'L_IDLE_STATE'. If the local backend, by command from the user, sends another writeset to the replication manager ('MSG_NEW_TXN' in the figure), the above loop is repeated. This goes on until the user disconnects the local backend. At that

time, the local backend sends a a 'MSG_CLOSING' and disconnects. The replication manager changes the state of the backend connection to 'L_DESTROY_STATE' to be recycled at a later stage.

The flow of events can be summarized graphically in Figure 5.4.

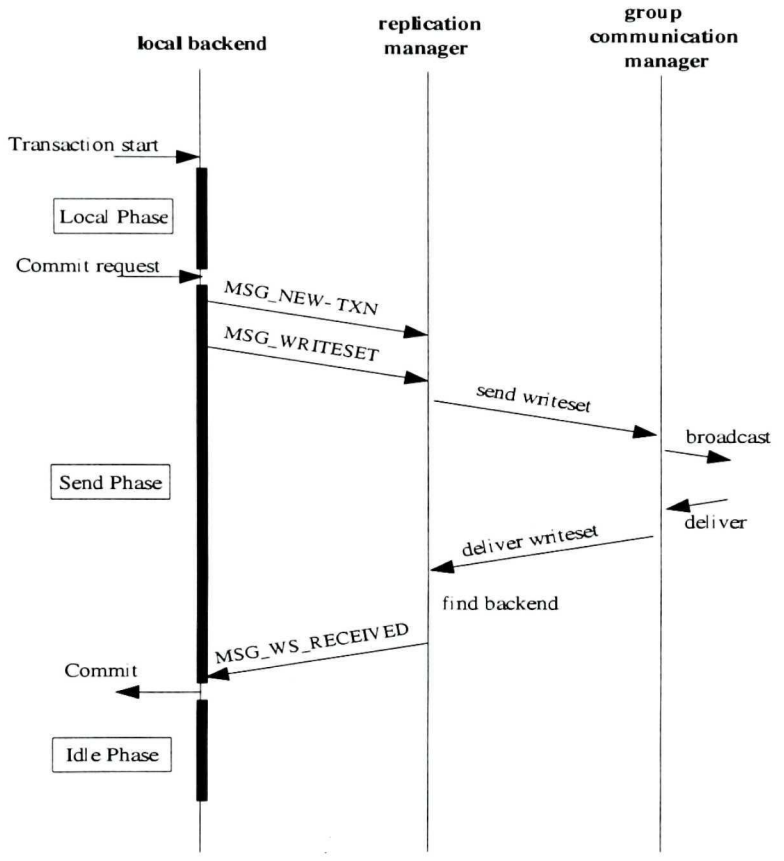


Figure 5.4: Flow of replication events at the master

At the slaves, the state machine is somewhat simpler, as there are only two states. The state of the remote backend is always 'R_FREE_STATE' unless it is busy processing a writeset, in which case its state is 'R_BUSY_STATE'. The remote backend advises the replication manager when it is finished processing a writeset by sending it a ready message. The state machine is illustrated in Figure 5.5.

The flow of events is illustrated in Figure 5.6.

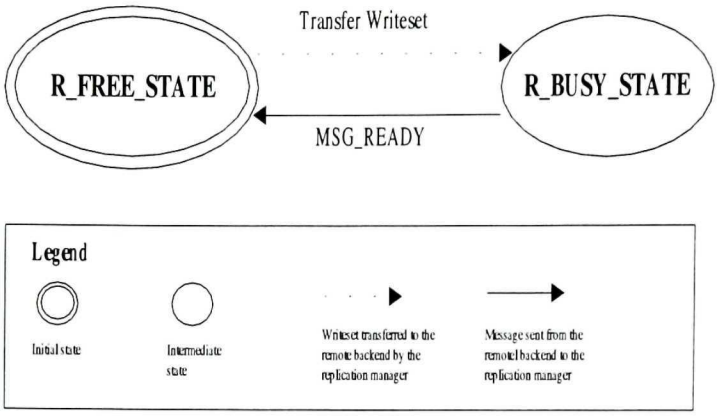


Figure 5.5: Replication state machine at the slaves

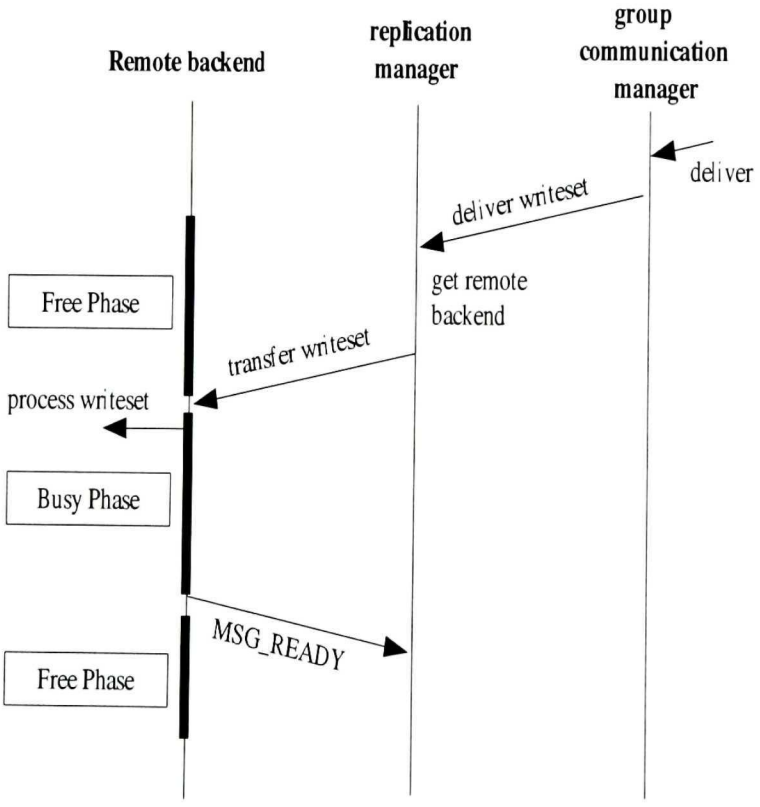


Figure 5.6: Flow of replication events at the slaves

5.3.2 Replication levels and the writeset format

Postgres-RFR uses two levels or replication, statement-level and tuple-level replication.

In the statement-level replication, the entire SQL statement is replicated, as a text string. The slaves have to apply the replicated string as though it was issued locally by a local client. This has the advantage of keeping the query processing module of PostgreSQL unaltered, since the statement will be copied to the writeset at a very high level. With Tuple-level replication, the modified tuples are sent to the slaves instead of the actual statement. On the master, the query must be executed to find all target tuples. These tuples are then collected and then sent to the slaves. The slaves use the indexes on the primary key to directly access the affected tuples and apply the changes. The relations must have a primary key defined. In the case of an insert, the entire new tuple is sent. In the case of an update, the primary key and the changes attributes are sent. In the case of a delete, only the primary keys are sent.

Postgres-RFR uses the good of both alternatives described above. IDU commands can be replicated in two ways, however utility commands are always replicated in a statement level manner. With an IDU command, the default replication mechanism is tuple level, unless the collected number of tuples exceeds a certain predefined number. If that number is exceeded, the tuples collected are discarded and the statement is included in the writeset, instead of the tuples.

The writeset functionality is taken from the Postgres-R implementation. Only a brief description is included here. For a detailed description, refer to [6]. The writeset is a message format that Postgres-RFR uses to propagate the updates from the master to the slaves. The master uses one writeset per transaction. The writeset is composed of fields that contain enough information to enable the slaves to recreate the transaction, and apply it on their local database copy. The group communication system handles all the communication between the master and the slaves, so writesets are also sent through these channels.

With statement-level replication, the writeset consists of a list of query strings and a list of table names. In this case, a writeset contents consist of character data. Table names are prepended to the respective query strings and sent to the slaves. This allows the remote transactions to access table names without parsing the query string. For tuple-level replication, tuple data have to be stored in the writeset. For insert queries, only the new tuple has to be stored. Delete queries only require the primary key values of the tuple to be deleted, and finally, update queries need the primary key and the new tuple to be stored.

5.3.3 Replication message format

The general message format is taken from the Postgres-R implementation, but since the flow of execution is different in Postgres-RFR, the set of message types needed differs from Postgres-R. Collecting and applying the writeset, both for tuple-level replication and statement-level replication, was taken from the PostgreSQL implementation. There are eight message types used to handle the master-slave replication protocol. They are summarized in the table below.

	Message type	Description
1	MSG_OPENING	A local backend always sends this message first if it wants to connect to the replication manager
2	MSG_WRITESET	This message represents the actual writeset sent by the local backend to the replication manager at the master, and by the replication manager to the remote backend at the slaves
3	MSG_NEW_TXN	A local backend sends this message to the replication manager if it has a new writeset to send
4	MSG_WS_RECEIVED	This message is sent by the replication manager to the local backend after receiving the writeset back from the group communication system. This is needed to allow the local backend to commit the transaction in question
5	MSG_CLOSING	A local backend sends this message to the replication manager when the user quits the client
6	MSG_READY	The remote backend sends this message to the replication manager after it finishes processing a writeset
7	MSG_PROTO_ERROR	This message format is included any time a message doesn't conform to an expected message format
8	MSG_INVAL	This is the default type for any newly created message container

5.4 Replication by command

So far, we focused on the replication mechanism, how the writeset is constructed and how the slaves apply these changes. In this section we will look at which of the vast set of Postgres-RFR commands is replicated.

As described above, Postgres-RFR commands could be divided into IDU commands and Utility commands. IDU commands are replicated in a straight forward way, either by tuple-level replication or statement-level replication. However, we need to

pay more attention to utility commands. Utility commands will only be statement-level replicated. One of the reasons behind the complexity of utility commands is the fact that numerous commands require the OS's support. To explain the idea, consider the following example:

```
CREATE FUNCTION point(complex) RETURNS point
AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
LANGUAGE C;
```

This example creates a function that does type conversion between the user-defined type complex, and the internal type point. The function is implemented by a dynamically loaded object that was compiled from C source, by specifying the exact pathname to the shared object file on the local machine. For PostgreSQL to find a type conversion function automatically, the SQL function has to have the same name as the return type, and so overloading is unavoidable. The function name is overloaded by using the second form of the AS clause in the SQL definition above. The C declaration of the function could be:

```
Point * complex_to_point (Complex *z) {
Point *p;
p = (Point *) palloc(sizeof(Point));
p->x = z->x;
p->y = z->y;
return p;
}
```

The problem with replicating the above statement to the slaves is that we have no guarantee that the dynamically loaded object exists on all slaves; and if it existed, weather it would be in the same local path on all slaves. Obviously, this should not be expected. In cases like this, we have four solutions to handling these commands

as explained below. In the following, the term *allow* means that the command can be submitted locally to the particular site. *Disallow* means that the command can not be submitted. If the client tries to submit such a command, the system will return an error message. With *replicate*, we mean that the master will multicast the statement to the slaves after local execution.

1. Disallow on all sites i.e. disallow in a replicated environment.

2. Allow on the master, replicate and disallow on the slaves. When replicating the utility commands, the user has to make sure that the paths are *correct*, i.e. consistent on all sites.

3. Allow on the master and the slaves; do not replicate. The user (or administrator) has to call the same command, manually, on all sites to have uniformity of data.

4. Implement a 2PC (Chapter 2) based replication where all, or none, of the sites apply the command

The decision on which of the above solutions to use is not easy. In all of them there is a cost involved.

Disallowing these commands on all the sites is obviously a major restriction to the rich and competitive selection of PostgreSQL utility commands. As described in the following tables, a good number of utility commands would have to be disallowed in this case, a major hit to PostgreSQL powerful database engine.

A slightly better alternative is to allow them on the master but not on the slaves, and replicating them *optimistically*. That means that in the case of the example above the user has to take the strenuous task of verifying the uniformity of the paths on all sites. This involves connecting manually to all slave servers in the group.

The third alternative avoids the above manually-exhaustive problem, by disallowing the replication mechanism, and allowing these commands to be invoked locally

by the user. This means that the responsibility of database consistency is put in the hands of the user or the administrator. This solution is risky, as it should not be expected from users to perform this task, and even if we allow them to do so, inconsistency problems might arise. This solution also contradicts the automatic replication solution we set out to achieve at the beginning of our work on PostgreSQL.

The last alternative involves running a 2PC protocol to ensure that all or none of the sites apply the command before committing it on the master. Out of all the solutions presented, this one guarantees database consistency and removes the burden of the manual application of these commands by hand by the user. This solution, however, is costly in terms of inter-site communication. It is also complex to design and implement.

We propose to classify the utility commands into four sets. The first set included the commands that could be allowed and/or replicated in a straight forward way. The following table summarizes these commands.

	Command	Master	Slave	Replicate	Comment
1	BEGIN	Allow	Allow	No	These commands should be allowed on all sites. They don't need to be replicated as the database server on the slaves will add these to every replicated transaction.
2	COMMIT	//	//	//	
3	END	//	//	//	
4	ABORT	//	//	//	
5	ROLLBACK	//	//	//	
6	INSERT	Allow	Don't Allow	Yes	These are the principle commands of our replication protocol. They should only be allowed on the master and replicated.
7	DELETE	//	//	//	
8	UPDATE	//	//	//	
9	CREATE GROUP	//	//	//	These are *superuser* commands and should not be allowed on the slaves.They should therefore be allowed only on the master and sent to the slaves. This facilitates the work of the database administrators.
10	DROP GROUP	//	//	//	
11	ALTER GROUP	//	//	//	
12	CREATE USER	//	//	//	
13	DROP USER	//	//	//	
14	ALTER USER	//	//	//	
15	GRANT	//	//	//	These commands should be handled similarly to User and Group commands.
16	REVOKE	//	//	//	
17	TRUNCATE	//	//	//	This should be treated similarly to 'DELETE * FROM ...'
18	COMMENT	//	//	//	
19	SELECT	Allow	Allow	No	This command should be allowed on all sites, and not replicated.
20	DECLARE	//	//	//	These commands are mainly used for data retrieval, so they should be allowed on all sites, and not replicated.
21	FETCH	//	//	//	
22	MOVE	//	//	//	
23	CLOSE	//	//	//	

24	CURRENT_DATE	//	//	//	These are session information commands, and don't change any data. So, allowing them on all sites is safe.
25	CURRENT_TIME	//	//	//	
26	CURRENT_TIMESTAMP	//	//	//	
27	CURRENT_USER	//	//	//	
28	EXPLAIN	//	//	//	This is similar to the session commands.
29	LOCK	Allow	Don't Allow	No	Locks don't need to be propagated as only one process executes updates on the slaves in a FIFO order.
30	SET CONSTRAINTS	Allow	Allow	Yes	This is a within-transaction-block command, and should be replicated within the writeset.
31	SET TRANSACTION	Allow	Allow	No	There is no need to propagate the user-defined transaction isolation level as transactions are applied by one process on the slaves in a FIFO order.

The next set of utility commands involves commands that are not as straightforward as the above commands. We provide a *best-effort* replication solution to them, but there might be more options to consider. The following table summarizes these commands.

	Command	Master	Slave	Replicate	Comment
32	CREATE TABLE	Allow	Don't allow	Yes	In our replication protocol, replicated writesets are performed using a special process that has superuser privileges. This means that a regular user will not be able to access his/her own table(s) on sites other than the one where they created their table(s). To avoid this problem we chose to send a complimentary writeset that grants the right privileges to all on the slaves. This additional writeset is sent right after the first, and is applied by all slaves.
33	CREATE TABLE AS	//	//	//	
34	DROP TABLE	//	//	//	
35	ALTER TABLE	//	//	//	
36	CREATE VIEW	Allow	Don't allow	Yes	These are similar to the 'CREATE TABLE ...' commands.
37	DROP VIEW	//	//	//	
38	CREATE INDEX	//	//	//	Indexes don't need to be replicated, as they affect performance and not the result of an IDU command. Since indexes are expensive to maintain, one might think of not replicating all indexes on all sites. Instead, each slave might specialize in a specific set of queries, and have indexes that are only useful for those queries.
39	DROP INDEX	//	//	//	

40	CLUSTER	Allow	Don't allow	No	As far as we understand, this command might overwrite some information, such as granted privileges, so we don't think that it is safe to allow users to cluster tables on specific indexes. To preserve user privileges, this command should only be allowed on one site, the master.
41	SET	Allow	Allow	No	Setting local specific variables should be allowed on the slaves for user's convenience, and so should not be replicated.
42	RESET	//	//	//	
43	SHOW	//	//	//	
44	VACUUM	Allow	?	Yes	This command cleans out old data that is no longer needed. As far as we understand, this affects performance (queries are faster after a vacuum) and not the result of an IDU command. Therefore, we could decide individually on each site when to vacuum. On the other hand, vacuum deletes old data, and might conflict with other updating transactions. Allowing users to execute this command at random on the slaves might lead to conflicts.
45 (1)	COPY COPY TO	Allow	Allow	No	This is basically a read operation.

The next set of utility commands either allow an absolute path as input, or depend on other utility commands that allow that as their input. We disallowed them in our implementation. The following table summarizes these commands.

	Command	Master	Slave	Replicate	Comment
45 (2)	COPY COPY FROM	?	?	?	An absolute path is required with this command
46	CREATE FUNCTION	//	//	//	When creating a function, the user has the option of providing an absolute path to it.
47	DROP FUNCTION	//	//	//	
48	CREATE LANGUAGE	//	//	//	In creating a language, a call handler is required, which should have been created using the 'CREATE FUNCTION' command.
49	DROP LANGUAGE	//	//	//	
50	CREATE TRIGGER	//	//	//	This utility command might depend on functions created using the 'CREATE FUNCTION' command.
51	DROP TRIGGER	//	//	//	
52	CREATE TYPE	//	//	//	A similar argument to 'CREATE TRIGGER' applies here.
53	DROP TYPE	//	//	//	
54	CREATE AGGREGATE	//	//	//	A similar argument to 'CREATE LANGUAGE' applies here.
55	DROP AGGREGATE	//	//	//	
56	CREATE OPERATOR	//	//	//	A similar argument to 'CREATE LANGUAGE' applies here.
57	DROP OPERATOR	//	//	//	
58	LOAD	//	//	//	A similar argument to 'COPY FROM' applies here.

The last set of utility commands are the troublesome cases. We are simply not certain how to handle them. The following table summarizes these commands.

	Command	Master	Slave	Replicate	Comment
59	CREATE DATABASE	?	?	?	Users should not be adding and deleting databases in a replicated environment. Databases should be created before launching the system in replicated environment, i.e. in standalone mode.
60	DROP DATABASE	//	//	//	
61	SELECT INTO	//	//	//	This command is similar to an insert command, and therefore should be replicated. It also reads data. Slaves might read different values than the master since transactions start later on the slaves.
62	CREATE RULE	//	//	//	Rules can also fire upon select statements. It is not clear if they should be allowed on the slaves.
63	DROP RULE	//	//	//	
64	CREATE SEQUENCE	//	//	//	Sequence numbers should be handled with care in a replicated environment. Unless different sequence numbers are allocated for different sites, this commands should be disallowed. However, we believe this is an inconvenience.
65	DROP SEQUENCE	//	//	//	
66	LISTEN	//	//	//	These commands can be combined with rules that are triggered by table updates to notify frontends. It is not clear how these commands should be handled in a replicated environment.
67	UNLISTEN	//	//	//	
68	NOTIFY	//	//	//	
69	REINDEX	//	//	//	This command can be used to recreate all system indexes of a specified database, and as such should be handled with great care.

Chapter 6

Failover

6.1 Introduction

Failover, in general terms, is the mechanism by which an entity, such as a site, *takes over* a *task(s)* from another entity in the case of an *event(s)*. In this chapter, this definition will be adopted for the specific task of the master of the group, the event will be *system configuration change* (SCC), i.e. a change in the site constituents of the group, and the take over will be the *election mechanism* of the new master if the old master failed. A system configuration change occurs in the event of a crash of a site(s), the deliberate withdrawal of a site(s), the recovery of a site(s) after failure, the introduction of a new site(s), network partition, and network merge. In the case of this event, one of the following scenarios should happen:

- The master remains the same
- the master *left the group*, i.e. either failed or was deliberately withdrawn from the group, and a new master must be chosen
- the master has not failed but a more powerful site has joined and should become master

In order for the failover to work properly, a new protocol has to be designed. We will refer to this protocol as the *Failover Protocol*. The protocol entails precise instructions for the behavior that all sites have to follow in the case of the SCC. The protocol is executed whenever the configuration of the system changes in order to determine which site in the new configuration is the master. The protocol might determine that the old master remains master or that a new master is chosen. After the execution of the protocol it is guaranteed that each site in the system has decided on the same site to be master. The failover protocol should be designed to be self-contained and no manual input from system administrators should take place.

In this chapter, we will try to address the above issues by giving details about the failover protocol. We will also cover implementation details. But first we need to look at the requirements of a failover protocol.

6.2 Failover Requirements

Failover is based on specific cornerstones that make up the 'work bench' of the protocol. These cornerstones include a *configuration file*, the *system state*, and the *election* of the new master.

6.2.1 The configuration file

The configuration file provides a *priority* listing of all possible sites that can ever be in the group. This file should provide a specific priority, or weight, for every one of these sites. This file represents a reference for all sites to consult in order to decide, based on the priority, on the new site to be the master in the case of the SCC. The configuration file is usually created by system administrators before launching the system. An identical copy of this file is stored at all sites, in a location known to the database server. The database server can either load the contents of this file at

startup, or at the occurrence of the SCC. This is illustrated graphically in Figure 6.1.

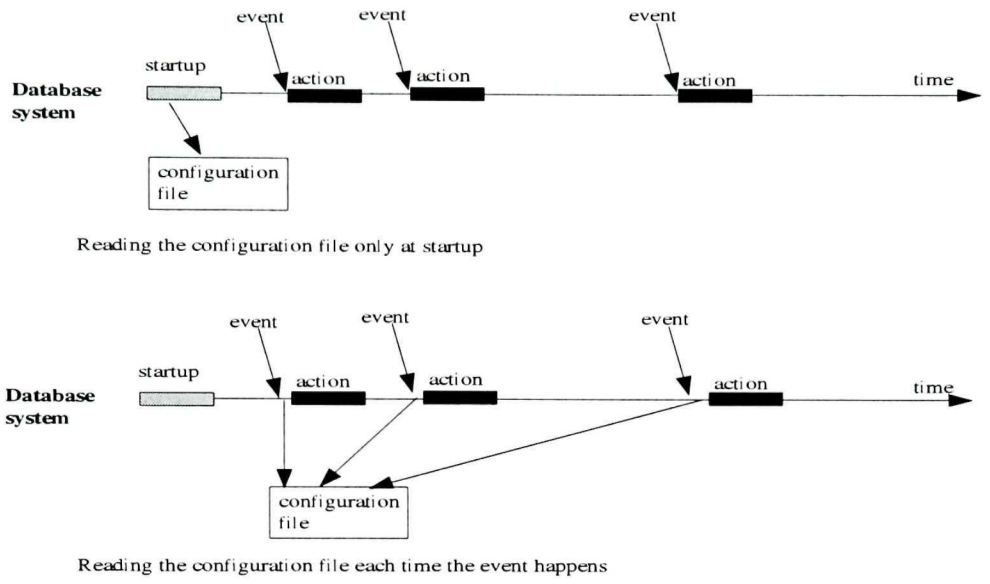


Figure 6.1: When to read the configuration file

The advantage of reading the configuration file at system startup is that the file is read only once, as opposed to many potential times as in the second case. However, the startup takes longer and there is also the chance that the SCC may never happen, which means that the failover mechanisms may never be needed. So even reading the file only once becomes a waste of time in this case. Reading the file each time also gives the system administrators the chance to adjust the file as they see fit, so that the new and modified file gets read the next time the configuration change happens.

6.2.2 The System State

On top of the configuration file requirement discussed in the previous section, the system sites require more knowledge about the state of the system before and after the occurrence of the SCC. The system state can be provided by the group communication system, or GCS (Chapter 4). In the case of the Spread system, for

example, the GCS maintains a *view*, i.e. snapshot, of the group at any time. In the case of a SCC, a *view change message* is delivered to the application on each site to alert them to the change. This view change message is composed of all sites that constitute the new configuration.

6.2.3 The election of a new master

In general, the sites that constitute the new configuration have to agree within each other if a new master has to be elected, or if the existing one remains as master of the new configuration. There are two general guidelines for this action. The sites can exchange messages to come to a decision on what to do, or they can take decisions based on accumulated knowledge.

The distributed election by message exchange

Distributed election protocols use message exchanges to come to a consensus, or conclusion. In the case of choosing a master for the group, the sites can run a distributed agreement to come to this decision. Distributed agreement protocols are similar to 2PC protocols (Chapter 2) in the sense that the protocol requires a coordinator and participants.

Election by accumulated knowledge

The members of the group can decide on the site that should take over simply by consulting a knowledge base. In the case of the master for example, sites can consult with the configuration file, described above, to decide on which of the other sites the administrator wants to take over the position of the new master. This knowledge base, as described above, can be acquired at system startup or each time it is needed. The individual sites, in this case, perform the failover protocol without any communication between them. The site that should assume the role of the master assumes it right away. That particular site assumes also that there is no other site

in the new configuration that will share with it the new acquired responsibility. All these decisions happen locally to the individual database servers. Election by accumulated knowledge requires two things to work properly. The configuration file and a snapshot of the new configuration. Sites compare the new configuration to the reference file, and can come to a uniform conclusion on the site that should become master. This is different from the distributed election based on communication, where individual sites only require the new configuration to come to the same conclusion.

6.3 Implementation details

The architecture of Postgres-RFR was not modified to accommodate the failover protocol. So the architecture described in Chapter 5 was preserved. Failover is mainly managed by the replication manager and the remote backend. To implement failover, new control structures and messages were required. Furthermore, a new user table was also added that contains the information about the different sites in the group. We refer to it here as the hosts knowledge base.

6.3.1 Configuration file

The satisfy the configuration file requirement described above, a new configuration file was introduced. This file is read at system startup, and never read again. The file contains information about all sites that the system administrators will ever introduce in the system. If new sites should be introduced, the system has to be shut down and restarted after the the configuration change.

Specifically, the configuration file contains the following information for each site:

1. The name of the site
2. The IP address of the site
3. The site priority

A high priority represents a site that the system administrators want to be the master of the group. A low priority means that the site should join and remain as a slave, unless all sites with higher priorities are down. It is up to the administrators of the database system to decide on a good priority for the sites, but things that have to be considered include the available resources to the site, i.e. speed, memory and hardware. The machine that is faster and has access to more memory than others should be given a high priority, so that it takes the job of the new master. It is also possible for two sites to have the same priority. In this case, our protocol will choose the one that appears first in the log. Figure 6.2 illustrates graphically an example of the use of the priority principle for a hypothetical system.

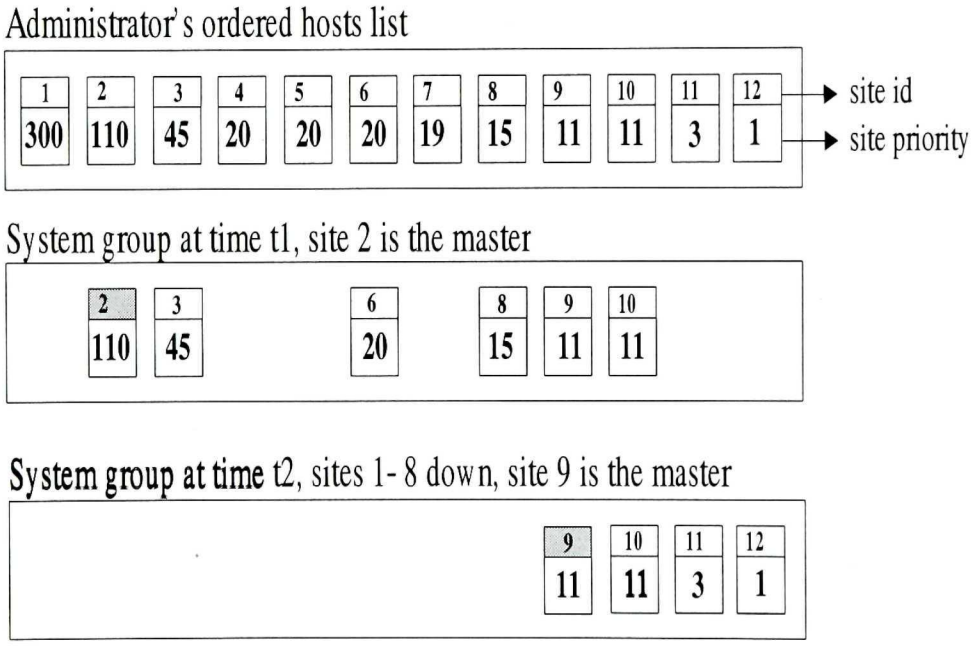


Figure 6.2: The use site priorities to decide on the master

In the example, the system is composed of a potential set of 12 hosts, each of which was assigned a priority as shown in the figure. These sites were numbered from 1 to 12 for illustration. At time t1, only sites 2, 3, 6, 8, 9 and 10 are up. The master of the group is therefore site 2, with priority 110. Some time later, sites 2,3,6 and 8

fail, and sites 11 and 12 are introduced. In this case, site 9 assumes the role of the master because it appeared first in the log.

6.3.2 Failover control structures

The purpose of the new control structures is to provide a control mechanism for sites crashing and re-entering the group. The main structures are two control lists, the *administrator list* and the *alive list*, as shown graphically in Figure 6.3. These lists are maintained by the replication manager in main memory.

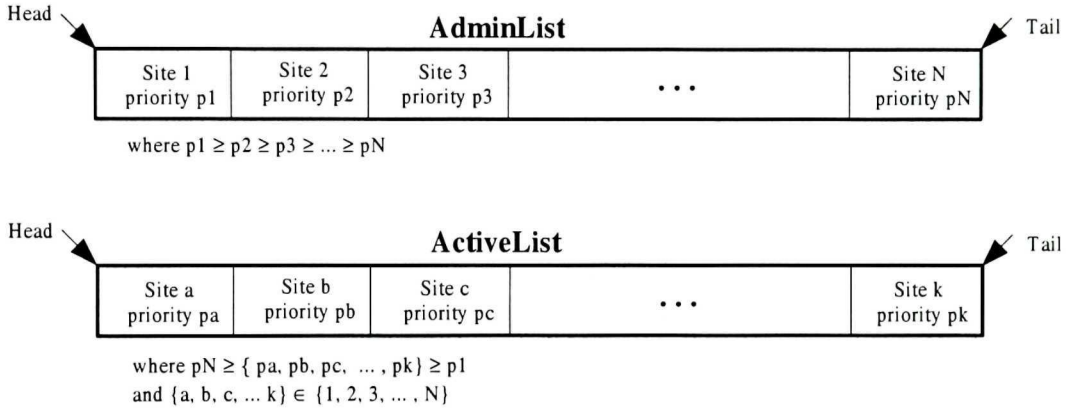


Figure 6.3: Failover control structures

The administrator list is a static list containing the information from the configuration file. The list is ordered by priority, with the site with the highest priority being at the head of the list. The ordering is a performance feature. It is intended to accelerate the traversing and the locating of sites with a specific priority. Figure 6.3 illustrates the administrator list graphically.

The alive, or active, list is the data structure that holds information about all the sites that are up and running. The set of sites in this list is a subset of the complete set of the administrator list. The alive list is not ordered, because it would be expensive to maintain a dynamic list, such as this one, in an ordered state. Besides, the ordering information is well provided by the administrator list. The alive list is updated each time a new view change is received. It represents a precise snapshot

of the configuration of the complete system. It is also the set of sites considered for choosing the master, each time the failover protocol is run.

6.3.3 System configuration knowledge

The system configuration knowledge is maintained by a user table, that we named *hosts*. This table is updated by the remote backend, on instructions from the replication manager. Database users can view this table, any time and at any site, to see which of the sites are up and running, and which of these sites is the master. This table has the following three columns:

1. The host name: this is the complete host name
2. The host ID: this is the ID of the machine
3. The host IP: this is the MAC address
4. The host type: this could be either 'M' for master, or 'S' for slave. At any time, there is only one 'M' in the table, and that letter denotes the only master of the group.

This table can be viewed by users through the standard PostgreSQL psql interface, 'psql', and the programming interface 'libpq'. To enable this functionality, two new *backslash* commands were added to the existing list of backslash commands of the PostgreSQL interactive interface. To view the hosts table, a user has to issue the following backslash commands:

- 'dh' for **d**isplay **h**osts. This displays a list of all active hosts in the system, as described above.
- 'sm' for **s**how **m**aster. This displays a detailed information about the master.

6.3.4 Failover Sequence of Events

The failover mechanism is triggered by the view change delivered by the group communication system. If the master of the group crashes, another site takes over that role. Also, if a site with a higher priority is (re)introduced in the group, the existing master has to step down from that position to allow the new site to assume the role of the master. But, what exactly is involved in the step of conversion? In this section, we intend to answer such question.

The view change message

Before doing so, we'll look in detail at the view change message. In Postgres-RFR, the view change message is a message sent by Spread to the group communication manager (Chapter 5). The latter processes that message and forms another message that the replication manager understands. This new message contains information about the new configuration.

This latter message is composed of two sections, the header and the message body. This is illustrated in Figure 6.4. The header of the replication protocol is the same as in the message format introduced in Chapter 5 for uniformity purposes, i.e. to keep message of the same format. The header is composed of the following fields:

- host_id: the host ID was given the special value 0, so that the replication manager can distinguish the view change message form the other received messages.
- trans_id: the transaction ID, just like the host ID, was also given the value 0. The combination of two zeros at the beginning marks the view change message uniquely.
- msg_type: the message type is 'MSG_VIEW_CHANGE'
- datalen: this is the total length, in bytes, of the data section of the message

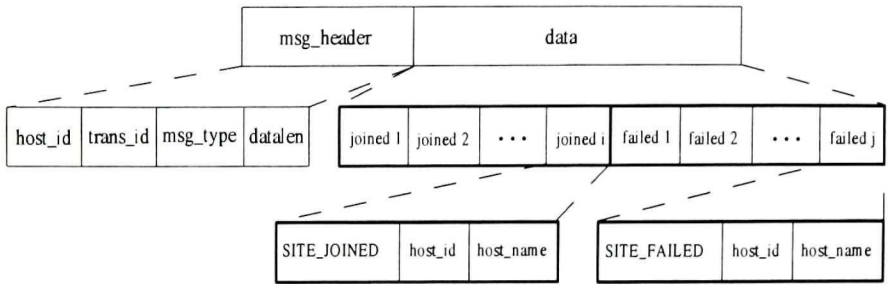


Figure 6.4: The view change message

The data part of the message is composed of two sections. The first lists the hosts, or sites, that joined the group. These are represented by the special 'SITE_JOINED' type. the second lists the failed sites , and they are represented by the 'SITE_FAILED' type. A host in the message is represented by its host name and its host ID.

The master state knowledge

Because the failover mechanism is designed for a dynamic environment, the individual database server has to be able to find out if its site is a master or a slave at any time. This knowledge is required by all the processes that take part in the architecture of Postgres-RFR. For instance, when a client requests an IDU command, the local backend has to decide weather the site is the master. Only then the command is allowed, otherwise the command is rejected and the client receives a corresponding error message. The shared memory contains a 'master-flag' that is true if the site is the master, and false if the site is a slave. We chose to put this information in a shared memory location at every site. All processes local to one site that need to know if their site is a master or a slave, can simply consult with this memory location. To avoid concurrency control, we allowed only one process to update this knowledge location, the remote backend. A new message type, 'MSG_MASTER_CHANGE' (Section 6.3.5), had to be designed that is sent by the replication manager, and instructs the remote backend to update that memory location. It is important to note the distinction between the 'master-flag' and the hosts table described above. The flag is a control structure that is internal to the

system, i.e. the users have no access to it. However, the hosts table is intended for display purposes, i.e. to provide a knowledge base that can readily be consulted by the system users.

Converting from a slave to a master or vice versa

After the view change message is delivered, the failover protocol has to decide if the master should be changed. If the view change message contains sites that (re)join the group, then one of those sites should also be considered as a potential master. In general, a site converts from a slave to a master in two scenarios:

1. When the master fails, so another slave has to assume that role
2. When a site that has a higher priority joins the system

In the first case, i.e. when the master fails, the slave with the highest priority takes over the role of the master. It changes its master-flag to 'true'. In the second case, the joining site sets its master-flag to 'true' and the master to 'false'.

In the second case, the conversion does not take place immediately after the view change since the joining site has to undergo recovery (as explained in Chapter 7). In this case, once the new site has completely recovered, it will multicast a special message to all sites. Only upon delivery of this message that the master conversion takes place. Apart from this view change message and this special message, there is no further communication between the sites keeping communication overhead low.

The Master change message

The master change message is a message sent by the replication manager to the remote backend to update the shared memory location that holds the knowledge about the site state (master or slave). This message has type 'MSG_MASTER_CHANGE'. Figure 6.5 illustrates this. The conversion type is either 'FROMMASTERTOSLAVE' or 'FROMSLAVETOMASTER'. The rest of the data is the new master information.

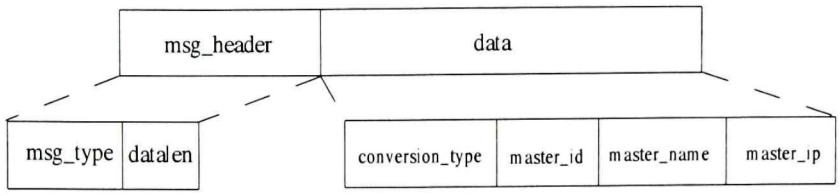


Figure 6.5: The master change message

Updating the client knowledge base

The client knowledge base is represented in our protocol by the hosts table. In order to update this table to reflect the status of all sites in the group, the replication manager has to send appropriate messages to the remote backend to instruct it to do that. This message, 'MSG_HOST_CHANGE' (Section 6.3.5), is sent by the replication manager to the remote backend to instruct it to update the hosts table with the information contained in the message. Updating this table can be either removing, adding or updating site entries. The update is simply to change a slave to a master, or a master to a slave.

6.3.5 Failover control messages

On top of control structures, new messages were required to handle the communication between the different processes in Postgres-RFR, namely the replication manager, the group communication system and the remote backend. Note that in our implementation, message exchanges take place in an asynchronous non-blocking fashion. That means that processes never block waiting. Instead, buffered sockets are used to buffer incoming messages, and later on processed. In our protocol, we use three new message types to handle the failover mechanism. The following table summarizes these message types.

	Message type	Description
1	MSG_VIEW_CHANGE	This is the view change message that is sent by the group communication system to the replication manager. This constitute the trigger to our protocol.
2	MSG_MASTER_CHANGE	This message represents the instruction sent by the replication manager to the remote backend to change the master status of the site.
3	MSG_HOST_CHANGE	This message represents the instruction sent by the replication manager to the remote backend to add or change an entry in the hosts table.

6.3.6 Failover and active transactions

There are times where a site could be processing transactions, and at the same time it changes its state, i.e. from a master to a slave or vice versa. IDU commands, for example, are rejected on slaves, but accepted on the master. Since view change message can arrive at any time, the database system has to properly handle active transactions at the time of the change.

In the case where the site changes from a slave to master, the database can alert the users that are actively processing transactions on its database. This way, users can start sending commands that were not possible before the change, such as IDU commands. With our implementation, this change happens transparently to the users, and hence they are not alerted to the change.

In the case where the site changes from a master to a slave, the database system can do two things: it can also alert users that are actively connected to the database, but also has to rollback their uncommitted transactions if they contained commands that are not accepted on slaves, such as IDU commands. Although this is inconvenient to the users, the database system guarantees this way the consistency of its data. Users have to restart their transactions on the master if they need to make changes to the database. If their transactions do not contain updates to the

database, they can simply continue using the same site. With our implementation, the local backends will simply fail by crashing in the case that the transaction contains updates to the database, and users have to reconnect to the database to perform their transactions. The local backend will not fail otherwise, and users can finish their read-only transactions.

Chapter 7

Distributed recovery

7.1 Introduction

Distributed recovery is the mechanism that a database undergoes to bring its own copy of data to be consistent with the other database group members. Distributed recovery is different from cental recovery. Central recovery brings the local database back into a consistent state. After local recovery, the changes of all transactions committed before the crash are included in the database, the changes of all transactions that committed before the crash or that were active at the time of the crash are not reflected in the database. After central recovery, the internal state of the database is consistent. But it does not reflect the transactions that were committed in the system while the site were down. For that, you need distributed recovery. Central and distributed recovery are both necessary and complementary to achieve the above state.

There exists several options for distributed recovery. In a simple solution, the recovering site simply gets a current and complete copy of the database from another site, or a *peer site*. However, the database might be very large, and only a small amount of data might have changed. A more efficient solution is for the recovering site to receive only the updates of the transactions it missed during its downtime.

The peer site can collect such state by scanning its central log and retrieving the redo logs of all relevant transactions.

In [19] several strategies of how to transfer state to the recovering site are proposed. In our work, we follow a slightly different approach, namely one where the recovering site receives all the transactions it has missed from a peer site. In this chapter we will provide a detailed solution for this approach, and show how it is implemented.

7.1.1 Overview of distributed recovery

An general overview of the distributed recovery is as follows.

- A site is (re)introduced in the system. It joins as slave.
- It then undergoes central recovery using the UNDO/REDO central log information (Chapter 2).
- It then joins the group, and as a consequence receives a view change from the group communication system (Chapter 4).
- From the moment it joins the group, it starts receiving writesets sent from the master to all the slaves. It then starts buffering these writesets as long as it is in distributed recovery (Section ...).
- From the list of sites in the view change, it chooses a peer site to help in the distributed recovery process (Section ...).
- It then establishes a communication channel with this site (Section ...).
- It then extracts from its distributed recovery log the identifier of the last transaction it has processed prior to failing. In the case of a new site, this identifier will be 0 (Section ...).
- It then proceeds to send this identifier to the peer site (Section ...).

- The peer site receives this transaction identifier, and extracts the updates of the transactions that occurred after that particular transaction (Section ...).
- The peer site then proceeds to send these updates to the recovering site (Section ...).
- The recovering site applies these updates (Section ...).
- After it has finished with applying the updates received from the peer site, it starts processing the writesets it has been buffering thus far (Section ...).
- When it has finished with the buffered writesets, it broadcasts a message telling the other sites that it is now ready to share in the load of the database cluster (Section ...).
- All sites then undergo failover mechanisms to decide if the newly recovered site should be the new master, or if it should stay as a slave (Section ...).

During this recovery process the master of the group still process user transactions. All sites, including the recovering site, still receive the broadcasted transactions from the master. Therefore, the recovery process must be coordinated with the receiving of transactions. That is, for new transactions sent by the master, the recovering site, while it is still in recovery, either receives them from the peer site or it receives them from the master, stores them in an intermediate queue and applies them after the recovery process is finished.

Distributed recovery is based on important database concepts such as system-wide transaction identificaation and logging. In this chapter, we present the fundamental concepts of distributed recovery. We will also present implementation details about Postgres-RFR distributed recovery protocol.

7.2 System-wide Transaction Identification

In a localized and central database system, every transaction that is executed is assigned a *local identification identifier*. This identifier is usually a unique number. When we look at a replicated database system, these locally created identifier numbers are no longer unique. If any system wide transaction control is required, a functionality to uniquely identify transactions has to be integrated. System-wide transaction identifier, or Global Transaction Identifier (GTI), refers to such functionality.

7.2.1 The UD properties

Any solution to the GTI problem has to satisfy two database properties: uniformity and durability. We will denote them as the UD properties in the rest of this paper. Let's look at these in more detail.

1. **Uniformity:** any GTI issued for a particular transaction has to be uniformly accepted and adopted by all servers in the group. Failure to do so, for any reason, by any server might jeopardize the consistency of the distributed database, hence the uniformity property.

2. **Durability:** this is of paramount importance for fault tolerant systems. A GTI is durable if it persists despite system failure. If a GTI does not persist in the case of failures, then the recovering site will not be able to globally identifier the last transaction(s) it performed before failing. It will only be able to locally identifier those transactions. Logging is the most common solution to the durability of data. We will therefore adopt logging as our primary means to ensure durability of the GTI.

The GTI problem is greatly simplified by the use of a master-slave protocol to replication, compared to solutions for update-everywhere replication. In a master-slave approach, the master is the only server that broadcasts the updates to the rest

of the group. It is only natural, therefore, that the master adopts the responsibility of issuing and governing the GTI. There exists, however, other alternatives to this conceptual design. In what follows, we present four designs to the GTI problem in a master-slave system. Handling the GTI problem in an update-everywhere alternative to replication is beyond the scope of this report.

7.2.2 GTI solutions

It is important to distinguish between the system-wide transaction identifier, i.e. GTI, and the local transaction identifier, i.e. LTI. The generation of the GTI of a transaction should be independent of the choice of LTI on the different sites. A transaction has exactly one GTI but might have different LTIs on the different sites of the replicated system. The generation of the GTI and the LGI are independent of each other. In this section, we present three alternatives to the GTI problem, namely piggybacking, using a distributed agreement protocol, and using the group communication system.

The piggyback solution

One of the options available to the master to enforce a unique GTI is to piggyback the newly issued GTI on the broadcasted writeset belonging to this transaction. message.

Uniformity is assured across all database servers since GTI is directly attached to the writeset and all servers receive the same writeset. Durability is the responsibility of the individual database servers. Once a writeset is received by a server, it has to ensure that the GTI is logged. This method has the desirable feature that there is no additional message overhead.

A distributed agreement protocol solution

The distributed agreement protocol is a technique to denote a message exchange between two or more processes to ensure that a particular decision is reached by all members of the group. This is similar to the 2PC protocol described in chapter 2 in the sense that the coordinator is the master, and the participants are the slaves. The agreement would be to reach a decision on a GTI suitable for all sites for a particular transaction. It can be originally *proposed* by the master, and the slaves have to vote if they accept it or not. This continues until all sites come to a uniform decision.

However, there is a high price for this solution: a communication cost. The number of messages exchanged between the master and the slaves is in the order of about three times the number of database servers.

Using the group communication system solution

Yet another alternative to the GTI problem is to take advantage of the semantics of the group communication system. As we saw in Chapter 4, the total order multicast guarantee of the GCS can be exploited to solve the issue of GTRs. Since the total order guarantees the total ordering of messages passed through the system, database servers could simply increment the GTI with every arriving new writeset. The GCS guarantees that even the sender, i.e. the master, will receive the writeset back from the GCS. So, no special treatment is required for the master. Every site, upon receiving the writeset from the GCS, assigns a new GTI to that particular transaction. In order to make it easy for the servers, incrementing the GTI by 1 is enough to guarantee the UD properties discussed above. The uniformity property is assured by using a uniform incremental value ('1' is good enough), and the durability property is assured by the individual recovery mechanism of the different sites.

This solution is attractive in many ways. It does not need any additional resources, it does not add any communication cost and it does not make the writeset functionality

any more complicated as we saw in the case of the piggybacking solution described above. It will also work for update-everywhere replication since the total order of messages also holds when more than one site sends messages.

However, it suffers a couple of drawbacks. The first is the total dependance on the underlying group communication semantics. That means any error in the group communication could automatically result in system-wide database inconsistencies. The second is the need for the recovering site to communicate with another site to discover the current value of the GTI.

7.3 Transaction information logging

In this section, we discuss how the GTI can be used for distributed recovery, what exactly should be logged by the sites, and how.

The GTI is of crucial importance in our distributed recovery protocol. It enables the recovering site to decide on an important issue, namely the GTI of the last update transaction processed before failing. It the informs a peer site about this GTI and the peer site will send the recovering site the updates of all transactions with higher GTI.

In central recovery (Chapter 2), undo and redo information for every transaction are logged, and for each terminated transaction either a commit or an abort record is logged. We call this information the *central log.* In the following we assume a no-force/steal strategy. Furthermore, each log-record contains the LTI. The distributed recovery process must perform a matching of LTI to GTI to find the redo logs of the transaction in the central log. An update message is formed that contains details about those transactions. These messages are then sent from the peer site to the recovering site.

Thus, distributed recovery requires the logging of two additional things: the GTI ans the matching between the GTI and LTI.

7.3.1 Logging Alternatives

In this section, we elaborate on the way the different database systems will log the GTRs, and any other pertinent transaction information.

Combining with the central logs

It is possible to combine the central logging with the distributed logging in one place. That is the central log can be modified to accommodate the extra transaction information needed to perform distributed recovery, namely the GTI information. For instance, each log record contains both GTI and LTI. This has the desirable advantage of centralizing recovery logging in one place, which makes it simpler to maintain and monitor. It also makes it simpler to implement. The existing implementation is simply expanded to accommodate the extra information.

However, this removes the modularity of the different recovery processes, namely central and distributed. If a database is to be run a stand-alone manner, distributed recovery mechanisms are of no use to it. This means that all the extra resources to manage distributed recovery requirements, i.e. memory and processing, are wasted, and hence performance is affected. In other words, it is desirable to have the choice between these different modules at system pre-installation configuration, and this can only be achieved by separating the two modules from one another.

Independent logging

As explained above, it is possible to separate the distributed recovery log from the central log, and this makes recovery more modular. In this case, the distributed recovery log must contain enough information to match GTI and LTI.

However, this will make the implementation more complex, and possibly redundant. This is because existing code from the cental recovery module might not be efficiently re-used, and new implementation has to be designed.

7.4 Distributed Recovery Steps

So far we analyzed the importance of GTI during normal processing. In this section, we'll look at the different steps in the distributed recovery itself. A site that has to recover, first establishes the needed communication with another site that will assist it in the process, and then at some point later, will be able to finish the process on its own. That point in time, we refer to it as the *self synchronization point.* Once it has accomplished that particular step, it becomes ready to share in the load balancing of the system, or it *actively joins* the group, i.e. it either becomes a slave and allows read-only transactions, or it becomes the new master (see Chapter 6).

7.4.1 Establishing a distributed recovery communication channel

The fundamental vehicle to replication and distributed recovery is message exchange between the database servers. There is a clear distinction in the message exchanges between replication and distributed recovery during normal processing. In the former, messages are sent by one server, the master, and received by all other servers, including the sender. In distributed recovery, however, messages can be sent by any server, and received by any other server, as explained below. Figure 7.1 illustrates this graphically.

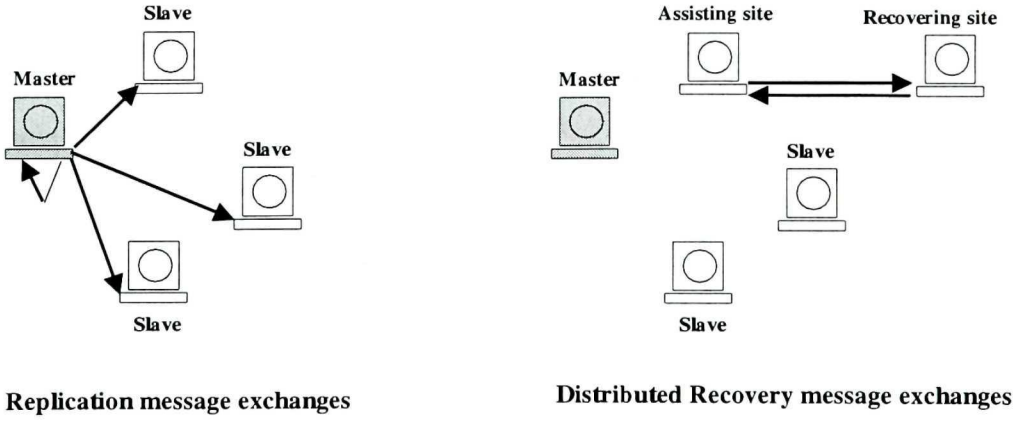


Figure 7.1: Difference in message exchanges between Replication and Distributed Recovery

A reliable message medium is of paramount importance for distributed recovery purposes. There are two methods that could be adopted for use to exchange messages in the distributed recovery protocol, a dedicated channel and a shared channel. In what follows, we analyze the advantages and drawbacks of each.

Using a dedicated communication channel

The dedicated communication channel should be totally isolated from the rest of the replication communication channels. It could be established between the recovering site and the peer site as soon as distributed recovery is deemed necessary, and abolished right after the process is finished. Figure 7.2 illustrates this idea graphically. This channel could be established to use TCP transmission protocol. We don't believe that UDP is a suitable choice here, because UDP does not guarantee delivery.

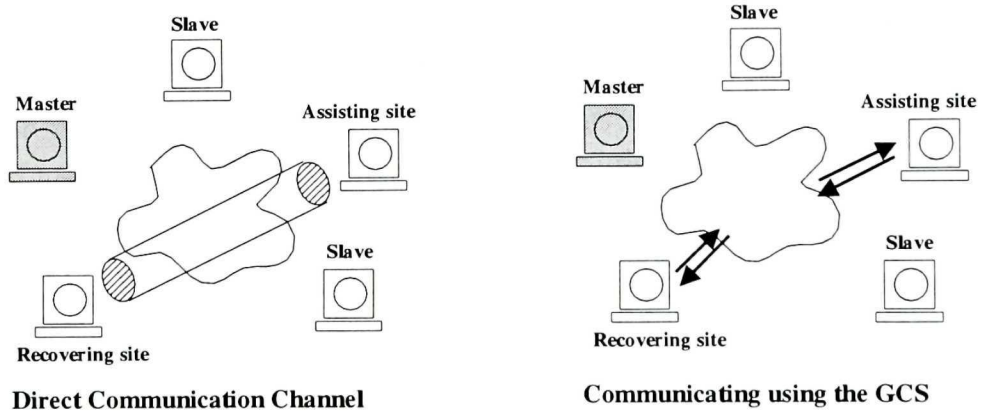


Figure 7.2: Communication options

The main advantage of using a dedicated communication channel is its independence of the replication communication channels. The dedicated channel will not get affected by failures, slowdown or congestion of the replication communication channels. A dedicated channel implies that distributed recovery messages are processed as soon as they are received by the recovering site. This, we believe, should enhance performance.

There are some concerns to distributed recovery over a dedicated communication channel. The first is a general concern to any communication channel between two computers over the internet: authentication. An important part of the establishment of this channel is a two-way authentication mechanism, where each of the two servers has to provide its identity as well as prove its legitimacy to the other. Another disadvantage to the dedicated channel design is the maintenance burden that accompanies such channels. This includes, on top of connection and disconnection management, tearing down of the channel, buffer management and required system resource management.

Using the replication communication channel

In this case, both replication and recovery protocols share the same channel, i.e. they use the group communication system (GCS). This is illustrated in Figure 7.2. In a query-intensive system, i.e. the ratio of updates to queries is relatively low, and a system where failures are not frequent, this solution is adequate. This is because the load on the GCS is relatively low. The underlying GCS channels, however, can be extensively used in the case of ann update-intensive system, where the ratio of updates to queries is relatively high. So, sharing in a channel that is already busy can act to the detriment of replication as well as distributed recovery.

The advantage of sharing the GCS channels is the ease to implement such a functionality. Since most GCS not only support multicast messages, but also point-to-point messages that can be used between the assisting site and the recovering site. There should be no need to implement other authentication mechanisms, as the GCS should handle this functionality.

However, this simplicity comes at the price of performance and speed. It is very desirable that the recovering site receives the recovery messages without any delays, so that it can finish with this phase as fast as possible, and start sharing in the load balancing of the entire database system. Sharing a communication channel means that the recovering site will have to screen between messages received for

recovery purposes, and other messages such as writesets. In the least case, it will have to process the headers of all messages to perform this task. This is different from the dedicated channel where all messages received on that channel are recovery messages.

7.4.2 Recovering site self synchronization

A crucial and delicate step in the recovery process is self synchronization of the recovering site. Self synchronization is the moment when a recovering site does not need anymore the help of its peer site, and can finish the distributed recovery process by itself. This moment occurs when the recovering site applies all the updates that were sent by the peer site, and has to switch over to applying the writesets received from the master, and were buffered up to this point. The recovering site should have been buffering the writestes received from the master when it was busy exchanging messages with the peer site.

There are three alternatives to design the synchronization step. It can be designed to use unbounded buffering, or to synchronize with the peer, or to use bounded buffering.

Synchronizing with unbounded buffering

One of the buffering alternatives available to the recovering site is to buffer without storage concerns the writesets sent by the master. As soon as the peer site finishes sending all the missed updates, the recovering site scans through the buffered writesets, looking for the last transaction it applied. It then starts applying the transactions that came after that one. That particular transaction provides the synchronization needed for the recovering site to start applying the master writesets. Obviously, an endless buffer is adequate for a system where updates are not frequent, but for a heavily accessed system, where the ratio of updates to queries is high, this option is not appropriate. This is because the recovering site will need

to buffer all these writesets, a process that can be expensive in terms of memory requirements, especially if the recovering process takes a long time.

Using recovery-peer synchronization protocol

The recovering site can avoid buffering until the peer site exhausted all its log entries. At that point, by message exchange, the recovering site starts buffering master writesets, but keeps receiving updates from the peer site. While doing so, it constantly monitors the two message streams looking for its synchronization point. On success, it switches over to applying the master updates and asks the peer site to stop sending updates. This option is complex and involves continuous message exchange between the two sites even after the peer site has no more updates to send.

Using a bounded buffer

A more feasible alternative that solves the problem of the unbounded buffer requirement is to use a bounded buffer that stores a predetermined number of master updates. Once the buffer is full, it should start dropping some updates to accommodate the new ones. The bounded buffer in this case could be implemented as a FIFO list, where the oldest update is dropped to make room for the newest update, when the buffer is full. Figure 7.3 illustrates the point. The recovering site has to look for its synchronization point within its bounded buffer after applying the updates from the peer site. Because it might have dropped updates from its buffer, the job of localizing the synchronization point is not straight forward. The risk here is that this synchronization transaction might not exist in its buffer, i.e. it was dropped. So for this option to work, the peer site must keep sending updates until the recovering site is able to pin-point its own synchronization transaction, after which, the services of the peer site are no longer required. The choice of the number of transactions to buffer here is of paramount importance. A too large a number implies wasted buffer space but guarantees the synchronization step. A too small number increase the risk of not finding a synchronization point but saves memory.

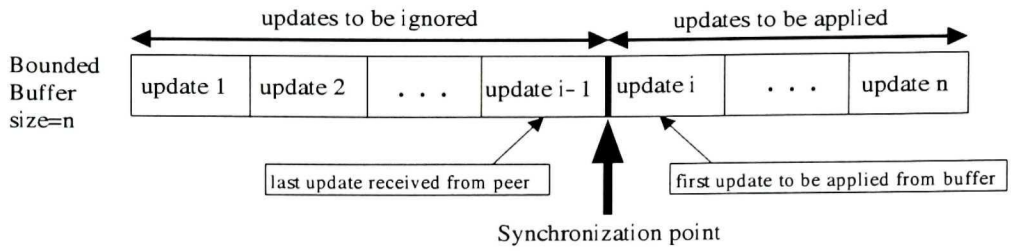


Figure 7.3: A bounded buffer approach to self-synchronization

The choice of the optimum size of the bounded buffer depends on many factors. The update load on the system being the dominant one. Other factors include network congestion, the speed of the peer site in sending the recovery messages, the speed of the recovering site in applying them and the available memory resources to the recovering site. For all these hard to control reasons, it is a hard job to estimate the right buffer size to use. To overcome this, a system could be designed so that the size of the buffer could be decided by the system administrator at system startup. The system administrator is someone who might have an idea about the load on the system as well as the network state, and might be in a position to take an educated decision on the parameter.

7.4.3 Switching from passive to active join

The introduction of a site into the system marks the beginning of its passive join. When a site is introduced, it has to bring its database to a consistent state. This phase is referred to as *passive join phase.* As soon as that particular site has successfully updated its database, it undergoes a transition from passive to an *active join phase.* A site starts sharing in the load balancing when it enters this phase.

Using the group communication services, a site receives a view change message from the group communication system that includes all the sites that are up and running at that time, including itself. The reception of that message marks the beginning of a passive join where the recovering site is not ready to process any updates from the

master. This phase continues until that site declares itself as active by broadcasting a relevant message to the group. A site enters an active join phase right after the delivery of such a message. Being an active site means that the site has successfully brought its database to the most current copy. During the passive phase, a site will perform all of the following jobs in the indicated order: Central recovery, Distributed recovery, Failover mechanisms. This is illustrated by Figure 7.4.

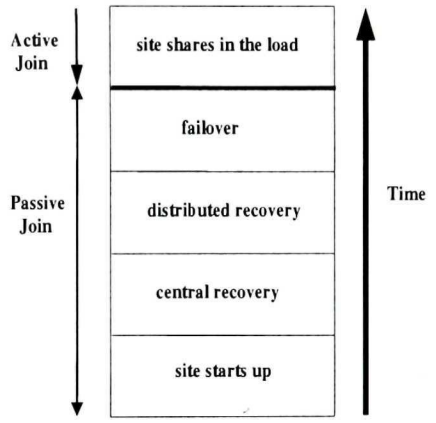


Figure 7.4: Switch from Passive to Active

As soon as a site is ready to share in the load balance, it is important that the other sites become aware of this. The fact that a recovering site has a consistent copy of the database (with respect to the other sites), makes it a potential site to assist other recovering sites. So, it is important to *advertise* this knowledge.

Broadcasting of a ready message

To advertise the fact that the recovering site is fully recovered, it can simply broadcast a 'I-am-ready' message at the end of its synchronization mechanism. Upon receiving this message, all sites update their knowledge base and consider the recovering site as active. This alternative minimizes the message overhead by keeping it to an absolute one broadcast message, that is sent at the very end of the recovery phase. Figure 7.5 illustrates this graphically.

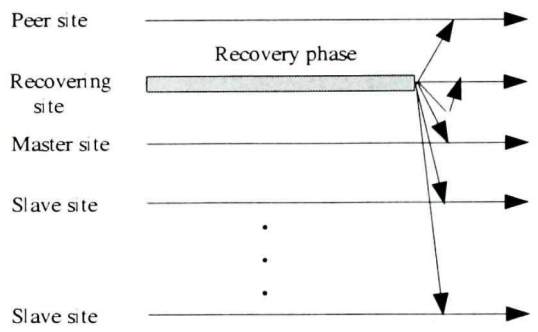


Figure 7.5: Ready state broadcasting by the recovering site

7.4.4 Election protocol of a peer site

Performing distributed recovery requires the assistance of an active site in the same group. There could be more than one other site that can fulfill this job, so a protocol to choose the *most adequate* has to be put in place.

The job of locating a peer host is not straight forward. This is because not every host that is active at the time when a recovering site is looking for assistance is capable of assisting in recovery. A host, for example, could be assisting another third host recover. A host should not assist more than one other database recover, unless of course, no other option is possible. In general, a particular host could be in one of the following states:

1. Assisting another host recover
2. Being assisted by another host in its recovery process
3. Looking for a peer host to assist in the recovery process
4. Waiting for a suitable host to be available for assistance

There are many mechanisms that can be used to choose this assisting, or peer, site. In here, we look at the most straight forward ones.

The Least-likely-to-be-the-next-master (LLNM) protocol

The most obvious choice for a this peer site is the one which is the least likely to be elected as the new master, should the existing master fail. The reason behind this choice is the idea that the master has already more work to do compared to the slaves, in terms of replication, so we don't want to burden it with more work, in terms of assisting in the recovery of newly introduced site.

The advantage of this protocol is straightforward. The master is relieved from the task of assisting a recovering site, while at the same time, it might be quite busy replicating data to the slaves. However, there are drawbacks to this protocol. A master is usually chosen on the basis of processing speed and available resources. So, the site that is the least likely to be elected as the next master will usually be the slowest, and/or the one with a reduced amount of resources, in terms of memory and storage. This implies that the distributed recovery process might take longer with the choice of this site than with other sites, especially in a highly replicated system, or if a large portion of the database needs to be sent to the recovering site. This drawback can be avoided by the next protocol: the Most-likely-to-be-the-next-master (MLNM) protocol.

The Most-likely-to-be-the-next-master (MLNM) protocol

The drawbacks of the LLNM protocol can be avoided by choosing a peer site that is the most likely to be elected as the next master, should the existing master fail. We abbreviate this protocol as the MLNM protocol.

The most forward advantage of this protocol is the fact that the peer site will be chosen among the fastest machines in the group, or the one that has a access to relatively decent resources. This implies that the distributed recovery process should theoretically take less time than with the LLNM protocol. However, in a high failure environment, we have no guarantee that a site chosen using this protocol will not be required to take the master role, should the existing master fail in the middle

of the distributed recovery process. Therefore, preventive measures should be put in place to account for this eventuality. This could be done either by temporarily removing this site from the list of potential future Masters, or lowering its election likelihood for the position of future master. This, evidently, calls for complex control and implementation.

The tie-breaker protocol

In all the above protocols, one can be faced with the case where two, or more, machines can fulfill the requirements of a peer site. This calls for a tie-breaker protocol.

As discussed in Chapter 6, the choice of the master is totally at the disposition of the people administering and maintaining the group. A master is chosen on a priority scale that the system administrators draft at the launch of the system. These priorities that dictate the order of choosing the master takes many forms, the most common is a configuration file. Two machines could possibly end up being assigned the same priority by the system administrator, either willingly or inadvertently. It is this scenario that calls for the tie-breaker protocol.

A straight forward approach to the tie-breaker situation is a random pick. Assuming a choice is to be made among more than one machine, one of them is chosen randomly. Another approach would be to choose the one that appears first in the configuration file (Chapter 6). Note that one can think of numerous tie-breaker protocols for this scenario. however, assuming that the priorities assigned to the individual machines are accurate, the performance of the general distributed recovery protocol should not be significantly affected by choosing one or another.

7.4.5 More than one site is recovering

A site that is recovering is not necessarily the only one performing that particular task. In a high failure system, more than one site could be undergoing a distributed

recovery. There are two concerns around this topic. Should this scenario be allowed to happen?, and if yes, should a peer site be allowed to assist more than one recovering site at the same time?

It should be an easy task for the system administrators to find out when a site is finished its distributed recovery process. All is required is to attempt to run a client on the recovering machine, and depending on the implementation, the database system can choose to allow client connections while in distributed recovery. So, theoretically, the distributed recovery system can be designed not to allow multiple distributed recovery processes to take place at the same time. This could be achieved, either by refusing a server startup, or leaving the control to the system administrators to leave enough delay between introductions of machines into the system.

7.4.6 Sending the missed transactions to the recovering site

The peer site, after receiving the last global transaction number from the recovering site, extracts all the updates that the recovering site has missed. It does this by consulting its distributed recovery log, and getting all the local transaction numbers that it applied after the received transaction. The peer host then proceeds to consulting its central recovery log to construct the message that will contain all the information required by the recovering site to bring its database up to date. There are two ways available for a peer site to construct this message.

1. It can use the write set functionality of the replication module
2. It can send the portion of the log as a data stream, and the recovering site should do the necessary parsing

Let's look at these two options in detail.

In the first case, the peer site has to form a write set, composed of all the updates of committed transactions. This list of write sets will have to be converted to a

byte stream and sent over to the recovering site using the communication channel established beforehand. The recovering site will have to reconstruct the writesets from the stream, and apply the updates using the replication functionality, using the same mechanisms as a slave when receiving writesets from the master during normal processing

In the second option, the peer site converts the relevant information in its local log into a byte stream right away without the need for the replication system. Upon receiving this stream, the recovering site will use the log interface to extract and apply the transactions in the stream. In here, the peer site might have to send a copy of its distributed recovery log if transactions are non-idempotent. An idempotent transaction is a transaction that if applied more than once, does not leave the database in an inconsistent state. The problem arises from the fact that a portion of a peer local log might contain records of transactions of no interest to a recovering site. So a recovering site, when coming across a record in the local log of its peer site, has to determine if it should apply the changes of that particular transaction, or if it simply should ignore it because it already applied the change. In the case that the database can apply the same transaction more than once without compromising consistency, a recovering site simply runs through the portion of the log and applies each transaction regardless if it already did so for that particular transaction. However, if that is not the case, a recovering site needs to receive both the central recovery log, as well as the distributed recovery log of its peer to be able to apply the transactions correctly.

7.4.7 Control transition: from recovery to failover

In a master-slave approach to replication, a site starts up as a slave and undergos recovery mechanisms. Because it is possible for that site to be the new designated master, either by the system administrators or by its default settings, a mechanism should be devised to enable that site to take over the position of the new master. We discussed this step in Chapter 6. The transition from recovery mechanisms to

failover should take place as soon as the recovery step is finished. This is because a newly introduced site that is intended to be the new master should assume that position as soon as possible. If the newly introduced site is supposed to stay as a slave, then nothing should be done.

7.5 Implementation details

In our implementation, we have chosen the following solutions as described in the previous section:

- The distributed recovery log and the central log are separated from each other. The distributed log keeps the GTI and their matching LTI.
- The recovering site uses a dedicated channel to communicate with the peer site, for the purpose of getting the updates it missed while it was down.
- The unbounded buffer alternative is used for the purpose of the self synchronization.
- The remote backend of the peer site generates writesets and sends them to the recovering site.

7.5.1 Postgres-RFR modified architecture

The architecture of Postgres-RFR, described in chapter 6 had to be modified to accommodate for the distributed recovery protocol. Mainly, a new type of backend used at the peer site to extract the data that must be transferred to the recovering site, has been added to the architecture. This new backend was given the name of *recovery backend.* The new architecture is outlined in Figure 7.6. In the figure, the remote backend is also shown. These two backends are created at system startup and are kept alive as long as the database server is up and running.

7.5.2 The distributed recovery protocol

The recovery backend has one main task, to get the data from the local log and send then to the replication manager. The replication manager calls on this backend when the site is required to assist another recovering site. The replication manager, in this case, instructs the recovery assistant to extract the missed transactions so that it

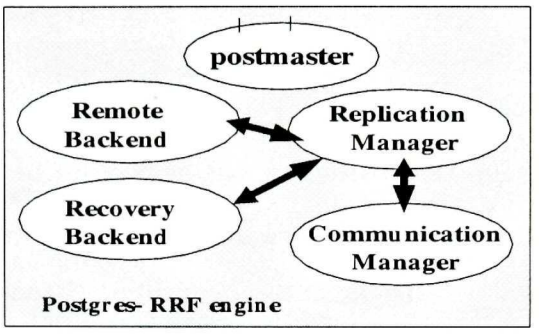


Figure 7.6: The modified architecture of Postgres-RFR

can send them to the recovering site. The interaction between the recovering site R and the peer site P, during recovery, is described below, with figures 7.7 and 7.8 to illustrate the protocol graphically. It is important to keep in mind that throughout the whole recovery process, the replication manager of the recovering site R stores all incoming writesets received from the master site in a buffer.

1. The recovering site joins the system
2. The replication manager of the recovering site R extracts GTI_{last}, the GTI of the last transaction that was processed before the crash by scanning its distributed recovery log.
3. It then tries to locate a peer site P to assist in the recovery.
4. It requests a connection directly to the replication manager of the peer site P.
5. Upon success, it sends the GTR_{last} to the peer site P, in the form of a 'MSG_LAST_TXN' (see below)
6. The peer site P, upon accepting the connection request and receiving the 'MSG_LAST_TXN', looks in its logs and extracts the LTIs of all transactions whose GTI is higher than GTI_{last}.
7. It sends this list L of LTIs to the recovery backend.

8. For each element in the list L, the following steps are performed:

   8a. The recovery backend extracts the corresponding transaction from the central logs, and forms a *recovery writeset*

   8b. It then sends these writesets, one by one, to the replication manager

   8c. The replication manager forwards the recovery writesets to the recovering site R, through the direct communication channel

   8d. The replication manager of the recovering site R, once it receives the recovery writeset, forwards it to the replication backend for processing

   8e. Upon success, the replication backend sends a special 'MSG_STEPACTION' message to be forwarded to the recovery assistant of the peer site P.

   8f. the replication manager simply forwards this message to to the peer site P.

   8g. The recovery assistant, upon receiving this message, forms the next recovery writeset, and starts the loop over again.

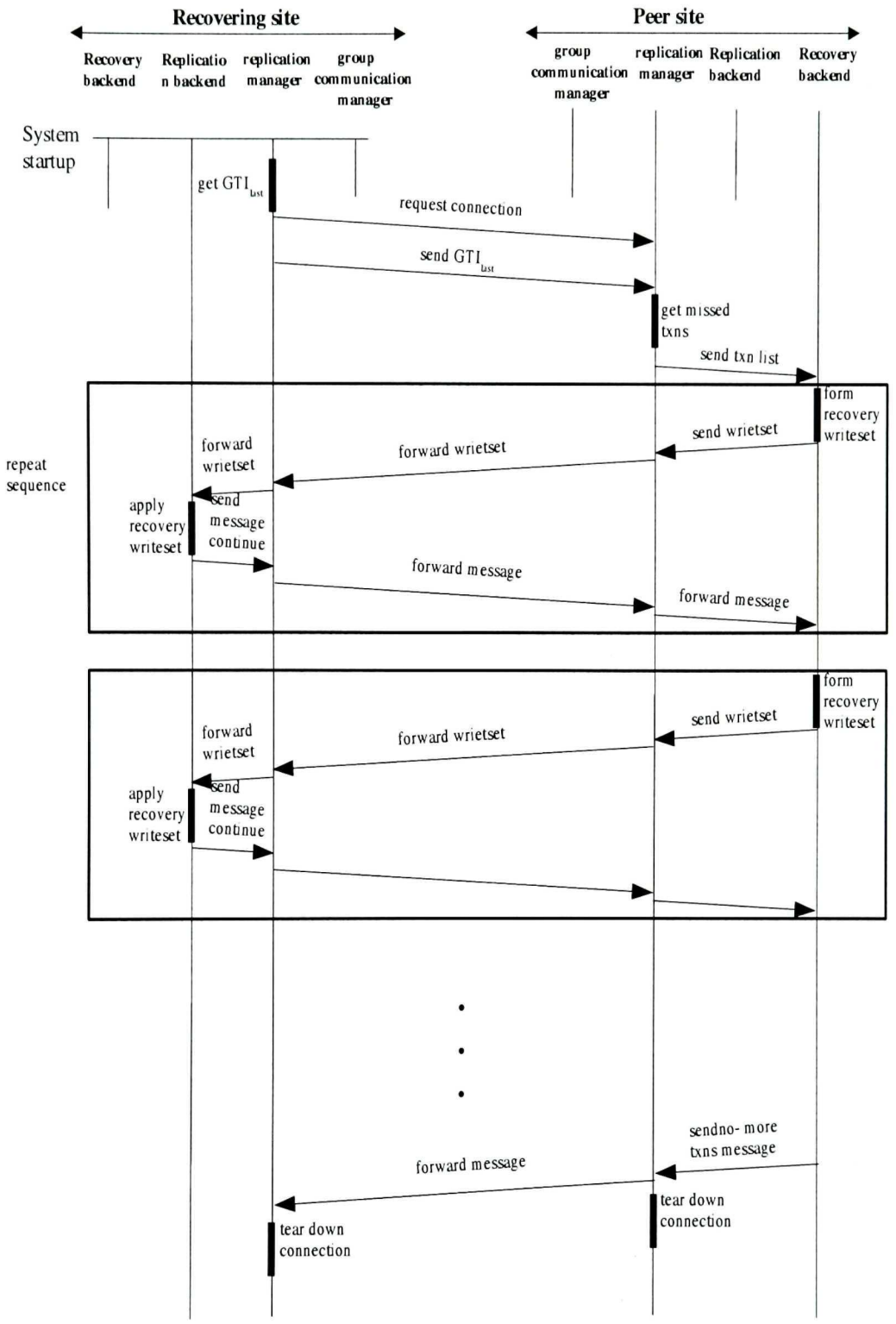
9. When there is no more transactions to be sent, the recovery assistant sends a 'MSG_NOMORETXNS' to the replication manager.

10. The replication manager of the peer site P forwards this message and immediately after, tears down the communication link and cleans away all the data structures used for the purpose of recovery.

11. When the replication manager of the recovering site R receives this message, it tears down the other end of the connection, and forwards the message to the replication backend for processing.

12. The replication manager of the recovering site R then proceeds to process the buffered writesets that were being buffered all along the above steps. So, one by one, it forwards these writesets to the replication backend to be applied.

13. When there is no more buffered writesets, it simply broadcast a 'MSG_SITEREADY' to all the sites



continued on next page ...

Figure 7.7: The distributed recovery protocol (1/2)

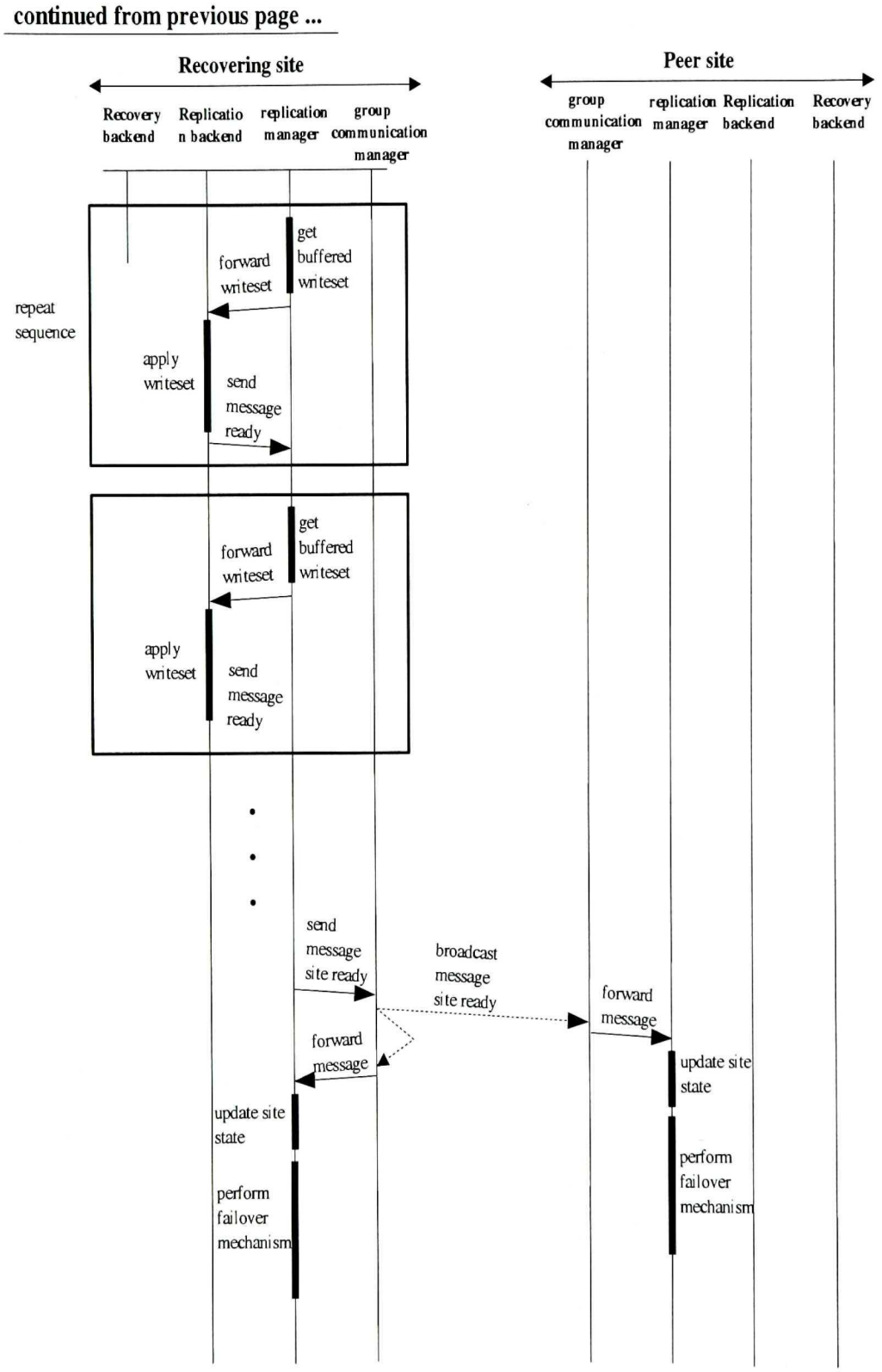


Figure 7.8: The distributed recovery protocol (2/2)

14. When any site, including the recovering site R, receives this message, they simply update that particular site from the recovering list (section 7.5.7) to the active list.

15. It might be desirable that R becomes the new master (e.g. it is the biggest machine in the group). Such takeover is handled by the failover mechanism explained in chapter 6. Hence, every site starts the failover mechanism to decide whether R should be the new master.

7.5.3 Postgres-RFR distributed recovery log

We adopted the approach where the distributed recovery log is separated from the central log. The distributed recovery log is managed by the replication manager. The following information is logged for every transaction that passes through the system, i.e. replicated.

1. The GTI: This is the globally unique identifier of the transaction. It is the same across all active sites.

2. The LTI: this is the locally unique identifier of the transaction. It is not necessarily unique across the entire system.

3. A dirty/final bit: this is a binary number that indicated that the record of the transaction in question is dirty or is a final record. A dirty record, as explained below, refers to a transaction that has not been committed yet on this site.

4. The backend identifier: the backend process, whether local or remote, that executed the transaction on the local copy of the database

5. The database identifier: the present database ID. Currently, it is assumed there is only one replicated database, but future versions will allow several replicated databases. In this case, such information is needed.

6. The identifier of the first record of the transaction in the central recovery log.

7. The identifier of the last record of the transaction in the central recovery log.

The location of the log is chosen to be the same as the central database logs, for simplicity reasons. Central logs (Chapter 2) of PostgreSQL are based on a tree structure, where nodes represent databases, and files represent transaction sets. However, in our implementation, we use only one file for logging. This should make it easier for the replication manager to perform the logging, yet the problem with this approach is that the file can get very large.

7.5.4 Logging sequence

There are eight instances where logging takes place in our protocol, 4 during normal processing and 4 during recovery. To satisfy the durability property, two records per transaction have to be logged, a dirty record and a final record. A dirty record is a record of a transaction not yet reflected on the database. A final record, on the other hand, represents a committed transaction. In four instances, a final record is logged. In four instances as well, a dirty record id logged. The following scenarios trigger a record in the log during normal processing:

1. On the master, when the replication manager receives a writeset from a local backend, it writes a dirty record

2. When the replication manager receives back the writeset from the group communication system at the master. This triggers a final record.

3. When the replication manager of the slave receives the broadcasted writeset. This triggers a dirty record.

4. At the slave, when the replication manager receives the message-ready from the replication backend, it writes a final record.

Figure 7.9 illustrates the logging instances during normal processing.

The following scenarios trigger a record in the log during distributed recovery:

5. When the replication manager of a recovering site receives a recovery writeset from the peer site, on the dedicated communication channel. This triggers a dirty record.

6. When the replication backend sends a 'MSG_STEP_ACTION' to the replication manager of a recovering site. This triggers a final record.

7. When the replication manager of a recovering site forwards a buffered writeset to the replication backend to process. This triggers a dirty record.

8. After processing a buffered writeset, the replication backend of a recovering site sends the message-ready to the replication manager of a recovering site. This triggers a final record.

Figure 7.10 illustrates the logging instances during recovery.

Dirty vs final record

In our protocol, a recovering site only considers the final record when searching for GTI_{last} in its distributed recovery log. The same applies for the peer site when searching its distributed recovery log to extract the GTIs of the transactions the recovering site missed while it was down. However, during normal processing, each site records two records for every processed transaction that contains update statements. In this section, we'll elaborate on the usefulness of the dirty record.

To explain this, we need to revisit the failure cases that can occur to a database during normal processing. In general, a database of a slave can fail at any moment, but the moments that are of particular interest for this section are the following:

1. A site can fail when the replication manager receives a writeset from the master, but before it transfers it to the remote backend for processing.

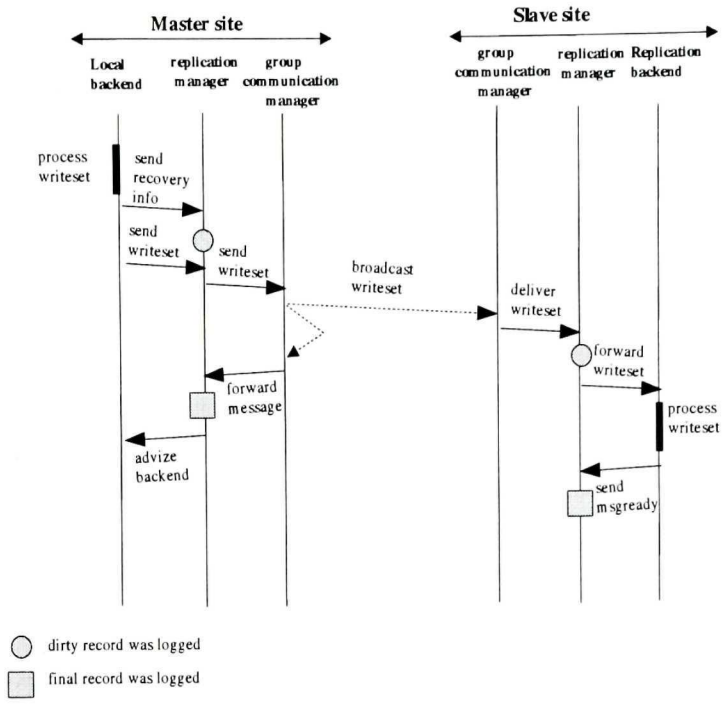


Figure 7.9: Logging instances during normal processing

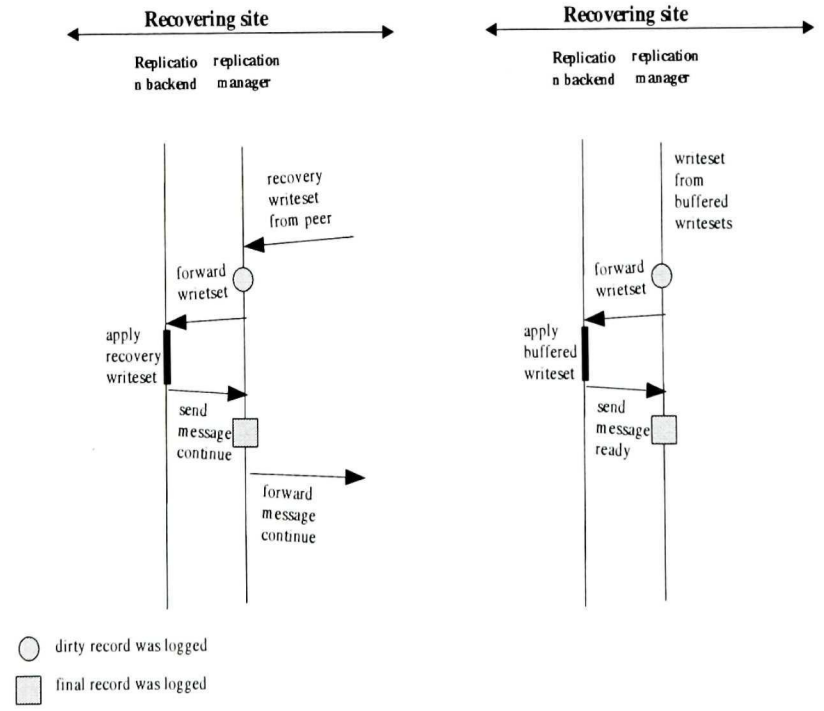


Figure 7.10: Logging instances during recovery

2. A site can fail after the replication manager transfers the writeset to the remote backend, but before the remote backend applies the changes of the writeset on the database

3. A site can fail after the remote backend finishes processing the changes, but before it alerts the replication manager about that, by sending it a confirmation.

4. A site can fail after the replication manager receives the confirmation from the remote backend.

When the replication manager of a slave receives a writeset from the master, it forces a dirty record in the distributed recovery log. This dirty record contains only a newly generated GTI, i.e. no corresponding LTI. If the database fails at this point, the distributed recovery log shows that it actually received a writeset from the master, whose changes might not have been reflected on the database. Upon recovery, the replication manager has no guarantee that the remote backend has processed this writeset, because a corresponding final record does not exist in the log. In this case, the replication manager can still ask for the updates of this particular GTI from a peer site, but has to verify that these updates have been applied on its database or not. It can do this as follows:

1. First, the recovering site has to get the updates of this particular GTI from the peer site, i.e. the GTI of the transaction for which the recovering site does not have a final record in its distributed recovery log.

1. It can then revisit its distributed log, and extract the LTI of the LGI_{last}, i.e. the corresponding LTI of the last GTI for which there is a final record.

2. It can then send this LTI to the remote backend, together with the received updates.

3. The remote backend can extract all the transactions processed on the database, whose LTI is greater than the received LTI.

4. The remote backend can then compare the extracted transactions to the transaction received from the replication manager, and decide if that transaction was processed.

5. The remote backend can then process the actual changes if they were not reflected on the database, otherwise, it has to ignore the transaction. If it decides to proceed to process the updates, it sends the corresponding transaction information, including the LTI, to the replication manager so that this latter logs the information in the distributed recovery log.

6. The distributed recovery mechanism continues as explained above.

The above protocol was not implemented. It is a difficult task to compare updates of two transactions to decide if they are identical or not. The above discussion explains the core reason behind the need for a dirty record. in summary, a dirty record provides extra information to the recovering database about transactions that were active when the site fails, and whose changes might not have been reflected on the database. It provides the starting point, i.e. the GTI, of these transactions to investigate them further. The rest of the failure cases listed above, as well failure case that can arise during recovery, follow a similar discussion.

7.5.5 Self synchronization of the recovering site

An important step in our protocol is the synchronization step where the recovering site finishes receiving updates from the peer site, and starts processing the buffered writesets. In here, we will elaborate on this.

In our implementation, a recovering site joins the group and immediately starts buffering, in an unbounded buffer, all the writesets received from the master. It then proceeds to extract the GTI_{last} from its distributed recovery log and sends it to the peer site. When the peer site receives the GTI_{last} from the recovering site, it extracts from its distributed recovery log, in one step, the GTIs of the transactions that are greater than the received GTI_{last}. The recovering site receives the updates

of those transactions, and applies them on the database. Then, it processes, in FIFO order, all the buffered writesets. Our implementation works as long as the peer site did not process anymore transactions after receiving the view change message from the group communication system concerning the new site, the recovering site, joining the group.

To elaborate on this subtle point, let's use an example. Suppose that the view change message v was delivered after all sites processed transaction number $T100$. Suppose also that the recovering site R has a GTI_{last} of $T90$. If during the time that the recovering site R was extracting this GTI_{last} from its distributed recovery log, a new writeset was multicasted by the master, then all sites will assign the value of $T101$ to the GTI of this transaction. Obviously, R will buffer this wrietset, and P will apply it on its local copy of the database and therefore will force a record in the distributed recovery log that has a GTI of $T101$. When P receives a GTI_{last} of $T90$, it will send to the recovering site the updates of $T91$ to $T101$ inclusive. R will apply these updates, and when finished will proceed to apply $T101$, again, because it was buffered in its unblounded buffer. This is a problem, as the recovering site will be applying the same transaction twice. Unfortunately, we don't account for this scenario in our protocol, so our protocol works only if no more transactions are processed by the system between the time a recovering site starts recovery and the time it finishes, by broadcasting a ready message (Section 7.5.7).

7.5.6 Distributed recovery messages

In our protocol, we use seven new message types to handle the recovery mechanism. The terms dirty and final are explained in section 7.5.4.

	Message type	Description
1	MSG_LAST_TXN	This is the first message sent by the recovering site to the peer site to start the recovery protocol
2	MSG_RECOVERY_INFO	This message is sent by either the local or remote backend to the replication manager to flush a dirty log
3	MSG_TXNLIST	The replication manager of the peer site sends this message to the recovery assistant to start forming the transactions
4	MSG_NOMORETXNS	This message represents the end of the recovery protocol
5	MSG_STEP_ACTION	Our protocol is implemented in a 'step' manner controlled by the recovering site. See Figure 7.7
6	MSG_XLOG_FAILURE	Failure to read the distributed log results in this message, either at the recovering or peer site
7	MSG_SITE_READY	A recovering site has to broadcast this message to all sites in the group at the end of recovery

7.5.7 The tasks of the replication manager in our protocol

In our implementation, the replication manager plays a central role in the distributed recovery algorithm. It has the role of coordinating the whole process from searching for a suitable peer to synchronizing the application of the missed updates.

In summary, the replication manager performs the following functions in the distributed recovery process:

1. Locate a suitable peer host to assist in the distributed recovery process
2. Establish a communication channel with a peer host
3. Localize the last global transaction and send it to the peer

4. Receive the missed updates from the peer
5. Buffer all incoming writeset messages sent from the master of the group
6. Transfer the missed updates to the remote backend to be applied on the database

Locating the peer host

The protocol of choosing a peer host relies on the idea that the ideal peer is a site not likely to be chosen as a master for as long as it is busy assisting another site recover. We can only make a best effort approach in regards to this scenario. However, if this case happens, that site should come to the rescue and become a master, even while assisting another site recover.

To satisfy the above condition, the peer site should be chosen to be the site that is active in the group, and has the lowest priority. Recall that the system administrators have to update the hosts configuration file before starting any server. This file contains all the site information as well as their priorities. Refer to the failover chapter for details about the system administrator's hosts configuration file.

It is important to note that the choice of a peer host by a recovering site, and the peer host realization of its new responsibilities, all happen without any message exchange. The way will do that is by having all sites maintain enough information about all sites in the system to be able to take informative decisions like these. That information is in the form of three management lists that all sites maintain. The lists are:

1. An administrator list: this is an ordered list of all sites that can possibly be introduced to the system. The ordering is according to priority, with the site with the highest priority is located at the head of the list.
2. A active list: this is an ordered list of all sites that are active at any time. See Chapter 6 for details.

3. A recovering list: this is the list of sites that are recovering at any time

At system startup, the replication manager reads the administrator list from the configuration file, the active list is formed with the help of the view change of the group communication system and the recovering list is started out as being empty. Figure 7.11 illustrates an example of an administrator list, an active list and a recovering list for a hypothetical system.

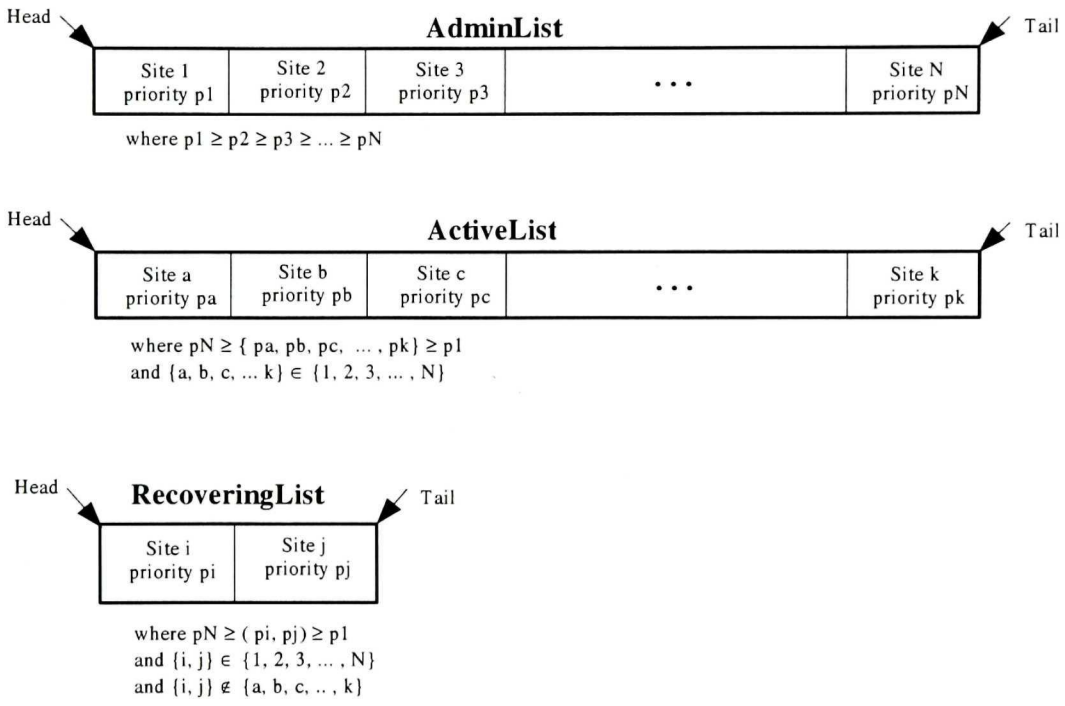


Figure 7.11: Hosts management lists

The only ordered list is the administrator list. The others don't need to be ordered. With the above data structures, locating a potential peer site could be performed by traversing the administrator's list from the tail, and checking for the first active host.

Establishing a dedicated communication channel

After deciding on a suitable peer host, a communication channel has to be established between them to transfer information. As described before, there are two methods we can use here for communication, using the group communication abstraction using a direct socket-enabled channel between the two hosts.

In the second case, a socket-enabled communication channel can provide for the direct and unhindered message transfer between the two sites. The replication manager of the recovering site has to establish this connection channel with the replication manager of the peer site.

Once a communication channel is established, a recovering site is ready to read its log files to decide on the updates it has missed during the downtime.

Localizing the GTI_{last} and sending it to the peer

This step relies on the recovery log that is maintained by the replication manager. At this stage, the recovery manager will extract the information about the GTI_{last}. Note that the GTIs are unique and consistent with all sites. This is because all messages delivered by the group communication system are totally ordered at all sites. So all sites will give the same GTI to the same message across the group. This is of fundamental importance to our protocol, as the synchronization mechanism is based on the fact that all updates are delivered in the same total order at all sites. A recovering site will look in its log and extract the GTI_{last}, it then proceeds to sending it to its peer site. The peer site will use this number to decide on which of the transactions the recovering site has missed while it was down.

There is a case when the replication manager can not find the GTI_{last}. This occurs with a new site that has never been introduced before. In our protocol and in this case, the GTI_{last} gets assigned the value of 0. The replication manager still sends this null value to the peer site in the same manner as with a non-null value. The peer site treats this null value just the same way as any other value. This means

that it will look in its logs and retrieve all the transactions with GTI's greater than 0. Obviously, all transactions in its logs will get retreived, as it should be case. Although this is the conceptual design of our protocol, this particular case was not tested.

Receiving the missed updates

The peer site sends the missed updates on the dedicated communication channel, so the replication manager receives them as soon as they are sent. Figure 7.7 above illustrates the message exchanges in our protocol. The replication manager forwards these writesets to the replication backend as soon as received, and monitors the progress. The replication manager checks the header of these messages, however, to see if the 'NO_MORE_TXNS' message is received. This last message represents the end of the message exchange between the two sites, and both tear down the connection with this particular message.

Buffering the master writesets

The replication manager has to buffer the updates broadcasted by the master of the group. Section 7.4.2 outlines the various buffering alternatives the replication manager can do.

In our implementation, we use an unlimited buffer approach. Even though this might not be the optimum solution, but our focus was on the correctness of the algorithm than its performance and efficiency. The buffers were designed to be a linked list, where writesets are buffered on a FIFO order, i.e. the head of the list is the first received writeset. Later, when it is time to apply these writesets, they will be processed in a FIFO order also. Note that buffering of writesets happen all along the recovery processing. This means that our protocol is not blocking, i.e. buffering of master writesets and peer site communication take place in parallel.

Broadcasting the ready message

We chose to broadcast the site-ready message to advertise the knowledge that the recovering site is ready to share in the load of the system. This means that even the recovering site is *not aware* of itself unless it receives this site-ready message. It only acts to clean out the recovery data structures after getting this back from the group communication system. This makes our code uniform across all sites, and no special handling was required for the recovering site.

Chapter 8

Conclusion and Future Work

In this thesis, we accomplished three tasks. We provided a master-slave solution to database replication, a failover mechanism to guarantee that there is always a master in the group, and a distributed recovery solution to enable a recovering site bring its own copy of the database to a consistent state with respect to the rest of the databases in the group. We PostgreSQL database and the Spread group communication system to carry out our implementations.

Our replication solution uses a synchronous approach, i.e. before transaction commit, to propagate the updates of transactions to the slaves. The database on the master collects the updates of the transaction in a special message. When the transaction is ready to commit, the master sends this message to the slaves. The slaves have to apply these changes on their local copies of the database. We paid particular attention to PostgreSQL utility commands in our replication solution. We provided a detailed breakdown of all these commands, and provided different replication alternatives to handle the rich capabilities of these commands.

Our replication work is centered around the underlying group communication system. We used Spread for this purpose because of its rich set of message guarantees and its group membership service. In particular, we exploit the total order semantic of Spread to guarantee the serializability of the multicasted transactions. The

group membership service of Spread is used to trigger distributed recovery of a newly introduced site.

Our failover solution guarantees that there is always a master in the group. This is important to our replication protocol, as it is based on a master-slave strategy. Our failover protocol relies on the accuracy of the system administrators in providing a detailed list of the sites that constitute the distributed system. We provided an interface for users to find out which of the sites of the group is the master, and to have a listing of all sites that are up and running.

Our distributed recovery solutions starts as soon as the recovering site joins the group. We had to extend the central and site-specific identification of transactions to a global system-wide level. We used the concept of global transaction identification to enable sites to reference transactions on a system-wide level. Basically, our protocol is a sequence of steps that the recovering site has to perform to bring its own database to a consistent state with the rest of the databases. First, it has to decide on the last transaction it processed before its downtime. It then chooses a site among the sites that constituted the view change message to assist in the distributed recovery. It then communicates with this site and gets from it all the transactions it missed while it was down. All along, the recovering site has to buffer the updates that are continuously being sent by the master. The recovering site processes these buffered updates, after finishing with the peer site.

Although our solutions have worked in most database scenarios, there are many venues for improvements and future work.

One improvement that can be applied to our replication solution is to provide a distributed agreement approach to replicating some of the utility commands of PostgreSQL. Many of these commands depend on parameters that are only known locally to a site, and an distributed agreement is required to ensure that all sites come to the same decision concerning these commands.

Our distributed recovery solution can be greatly improved. Among the many possible enhancements, we list the following:

- Exploit the distributed recovery log parameters, such as the dirty record, to handle some database failure cases.
- Enhance the structure of the distributed recovery log: make a tree-structure to the log, instead of a sequential one-file logging scheme.
- Provide a bounded buffer approach to buffering the updates that are received from the master.
- Combine the central log with the distributed log to centralize all recovery logging.
- provide a better management of the case when more than one site is recovering.

The next step of this project is to migrate our solutions to an update-everywhere approach to replication. In an update-everywhere replicated environment, every site can process updates from the users. With this, users don't have to locate the master and connect to it to perform updates. they can do that from any site in the group.

Another important, yet complex, improvement to add would be to handle network partitions. With our solutions, it is possible to have more than one master if there are network partitions. This is known to be a very complex distributed problem, which is the subject of active research.

Bibliography

[1] Gustavo Alonso, *Parallel and distributed databases course notes*, Department of Computer Science, ETH Zurich (2002), http://www.inf.ethz.ch/department/IS/iks/education/PDDBS/Summer02/foils/L6n.pdf.

[2] Y. Amir and J. Stanton, *The Spread wide area group communication system*, Johns Hopkins University, Center of Networking and Distributed Systems **Tech. Rep. 98-4** (1998).

[3] Yair Amir, Cristina Nita-Rotaru, Jonathan Stanton, and Gene Tsudik, *Scaling secure group communication systems: Beyond peer-to-peer*, Proc. of DISCEX3, Washington DC (2003).

[4] Yair Amir and Ciprian Tutu, *From total order to database replication*, ICDCS (2000).

[5] Todd A. Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool, *Replication, consistency, and practicality: Are these mutually exclusive?*, SIGMOD Conference (1998), 484–495.

[6] Win Bausch, *Integrating synchronous update-everywhere replication into the PostgreSQL database*, Master's thesis, ETH Zurich, 1999.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency control and recovery in database systems*, Addison-Wesley, Massachusetts, 1987.

[8] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangu, K. Kane, F. Schmuck, and M. Woods, *The ISIS System Manual, Version 2.1*, Dept. of computer science, Cornell University, September 1993.

[9] Luis-Felipe Cabrera and Jehan-Franois Pâris, Proceedings of the First Workshop on the Management of Replicated Data, IEEE Computer Society Press, 1990, ISBN 0-8186-2085-4, Houston, Texas, USA (November 8-9, 1990).

[10] PostgreSQL development group, *PostgreSQL project*, (2002), http://developer.postgreSQL.org/.

[11] D. Dolev and D. Malki, *The Transis approach to high availability cluster communication*, Communications of the ACM **39** (1996), no. 4, 64–70.

[12] Paul D Ezhilchelvan, Raimundo A Macdo, and Santosh K Shrivastava, *Newtop: A fault-tolerant group communication protocol*, International Conference on Distributed Computing Systems (1995).

[13] M. H. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of consensus with one faulty process*, Journal of the ACM **32(2)** (April 1985), 374–382.

[14] J. Gray, P. Helland, O. O'Neil, and D. Shasha, *The dangers of replication and a solution*, ACM SIGMOD (1996), 173–182.

[15] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi, *The performance of database replication with group multicast*, FTCS (1999), 158–165.

[16] IBM, *IBM database replication*, 2002, http://www-3.ibm.com/software/data/dpropr/pres9/index9.html.

[17] B. Kemme and G. Alonso, *A suite of database replication protocols based on group communication primitives*, Proc. of ICDCS, Amsterdam, Holland (1998).

[18] B. Kemme and G. Alonso, *Don't be lazy, be consistent: Postgres-R, a new way to implement database replication*, Proc. of the 26th International Conference on VLDB, Cairo, Egypt (September 2000).

[19] B. Kemme, A. Bartoli, and O. Babaouglo, *Online reconfiguration in replicated databases based on group communication*, Proc. of IEEE International Conference on Dependable Systems and Networks, Goreborg, Sweden (June 2001).

[20] Bettina Kemme and Gustavo Alonso, *Database replication based on group communication*, Technical Report No. 289, ETH Zrich, Departement of Computer Science (1998).

[21] Zachary Kurmas, *A survey of tradeoffs between guarantees in reliable multicast*, Technical report, College of Computing's systems, Georgia Tech (1998).

[22] L. Lamport, *Time, clocks and the ordering of events in a distributed system*, Communications of the ACM **21, 7** (1978), 558–565.

[23] Yao S. B. Lehman, P. L., *Efficient locking for concurrent operations on b-trees*, ACM Transactions on Database Systems **6, No. 4** (1981).

[24] L. E. Moser, Y. Amir, P.M. Melliar-Smith, and D. A. Agarwal, *Extended virtual synchrony*, Proc. of the ICDCS, Poznan, Polland (1994), 5665.

[25] L. E. Moser, P. M. Melliar-Smith, D. Agarwal, Y. Amir, R. K. Budhia, and C. A. Lingley-Papadopoulos, *Totem: A fault-tolerant multicast group communication system*, Communications of the ACM **39** (1996), no. 4, 54–63.

[26] Lotus Notes, *Lotus Notes database replication*, (2002), http://www.lotusnotes.com/Software.nsf/Home?OpenForm.

[27] ORACLE, *Oracle database replication*, (2002), http://www.orafaq.org/faqrepl.htm.

[28] M. Tamer Ozsu and Patrick Valduriez, *Principles of distributed database systems*, Prentice Hall, New Jersey, 1999.

[29] Esther Pacitti and Eric Simon, *Update propagation strategies to improve freshness in lazy master replicated databases*, VLDB Journal 8(3-4) (2000), 305–318.

[30] Fernando Pedone, Rachid Guerraoui, and Andr Schiper, *Exploiting atomic broadcast in replicated databases*, Euro-Par (1998), 513–520.

[31] R. V. Renesse, K. P. Birman, and S. Maffeis, *Horus: A flexible group communication system*, Communications of the ACM **39** (1996), no. 4, 76–83.

[32] Sybase, *Sybase database replication*, (2002), http://www.sybase.com/products/middleware/replicationserver.

[33] Brian Whetten, Todd Montgomery, and Simon Kaplan Dagstuhl, *A high performance totally ordered multicast protocol*, Seminar on Distributed Systems (1994).