POSTGRESQL-RR INTEGRATING RECOVERY INTO POSTGRESQL-R SYSTEM

WeiBin Liang

A Thesis
Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science

School of Computer Science

McGill University

Montreal,Quebec

2005-06-01

Your file Votre référence
ISBN: 978-0-494-22744-2
Our file Notre référence
ISBN: 978-0-494-22744-2

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Bettina Kemme, for the enthusiastic supervision and tremendous helps she gave to my master's research. I would also like to thank my colleagues, especially Shuqing Wu, Yi Lin, and Huaigu Wu, who are always willing and ready to give their supports to me. I also want to thank my family for their endless loves and supports.

ABSTRACT

Prototypes of replicated database management system have been designed and implemented to improve the reliability and availability. Although these systems are quite robust to failure, a recovery mechanism is still missing to bring a failed site back into the system. This thesis studies two different distributed recovery strategies and presents a hybrid distributed recovery algorithm that combines the two strategies. These recovery algorithms using different strategies are also integrated into PostgreSQL-RR system. This thesis also compares the cost in term of time associated with recoveries using different strategies, which gives a good foundation to the heuristic used in the hybrid distributed recovery algorithm to automatically select an optimal strategy to reduce the recovery time.

ABRÉGÉ

Les prototypes de systme de gestion de donnes copi ont t conus et ont t appliqus pour amliorer la fiabilit et la disponibilit. Bien que ces systmes soient tout fait robustes l'chec, un mcanisme de rtablissement est calme manquant pour rapporter un site rat dans le systme. Cette thse tudie deux stratgies de rtablissement distribues diffrentes et prsente un algorithme de rtablissement distribu hybride qui combine les deux stratgies. Ces algorithmes de rtablissement utilisant des stratgies diffrentes sont aussi intgres dans le systme de Postgresql-RR. Cette thse compare aussi le cot dans le terme de temps associ avec les rtablissements utilisant des stratgies diffrentes, qui donne une bonne fondation l'heuristique utilis dans l'algorithme de rtablissement distribu hybride automatiquement pour choisir une stratgie optimale pour rduire le temps de rtablissement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

Over the years, we have witnessed an increasing demand for scalable and highly reliable and available systems. One trend to address this issue is to replicate resources, which includes both computing resources and data. In the database field, we focus on data replication. We can replicate data at different computers and connect them into a cluster to handle workload collectively. In this thesis, we call the computers in the cluster *sites*. When the demand shoots up, we can scale up the system by simply adding more sites to the cluster. We can also distribute these data replicas physically to where the data is needed to provide fast local access. Furthermore, if one site fails, its workload can be taken over by another site that contains the same replicated data, providing fault-tolerance. The challenge is how to perform transactions on these replicated data. A *transaction* is a collection of operations that logically belong together. An operation can be either a *read* operation or a *write* operation. A read operation gets the current value of a data object in the database, while a write operation updates the value of a data object. A transaction can either commit or abort. If a transaction commits, all updates by the transaction must be reflected in the database. Otherwise, none of the updates by the transaction should be reflected in the database. This is referred to as transaction atomicity. Although concurrent transactions may access the same data, the execution of these transactions must have the same effects as if they were executed serially. This is referred to

as transaction isolation. When a transaction now accesses replicated data, it usually follows a read-one/write-all approach. With this approach, it reads data from only one replica. However, when it writes data, it must either perform the write on all replicas or perform it on one replica but then this replica must propagate the changed data to the other replicas to keep data consistent. Different replication approaches have been developed to satisfy the above transaction properties at different degrees. They can be categorized based on two parameters: (1) *where* write operations are performed; (2) *when* update information of a transaction is propagated to all replicas. In regard to (1), we distinguish two approaches. In the master-slave strategy, all write operations are performed at one site, the so called master site, while all other sites of the system, called slave sites, process read operations. After processing a write operation, the master propagates the changed data to the slaves in order to keep the replicas consistent. In contrast, using the update-everywhere strategy, write operations can be executed at any site of the replicated system, and every site is responsible to propagate the corresponding changes to all other sites. In regard to (2), with the eager (or synchronous) strategy, update information is propagated within the transaction boundary; while with the lazy (or asynchronous) strategy, update information is propagated any time after the commit of the transaction. In theory, an eager update-everywhere approach is preferred, because data consistency is maintained across the entire system and the workload can be evenly distributed to all sites even for an update intensive system. However, this comes with a price. It requires distributed concurrency control (e.g. 2-phase-locking) and an agreement

protocol (e.g. 2-phase-commit) in order to isolate concurrent updates across the entire system and maintain consistency between data replicas. Hence, it often poses too much overhead to the system and the system suffers a huge performance downgrade. Furthermore, a database system using this strategy does not scale well. So, although major database management system vendors, like Oracle, do provide users this replication strategy option [9], in practice, commercial applications that cry for high throughput and low response time opt for the master-slave approach. However, for a write intensive system, the master site becomes the performance bottleneck. Also, the system is inflexible because update transactions may only be submitted to a specific site. Then, how about the lazy update-everywhere approach? With this replication scheme, data may need to be reconciled for conflicting concurrent transactions. For example, transaction T_a updates data object X to 0 at site S_a and commits. Before the change is propagated to site S_b, transaction T_b updates X to 1 at S_b. After both changes are propagated to both sites, X has different values at these two sites, and hence X needs to be reconciled. As it is argued in Gray's paper [14], the reconciliation rate shoots up dramatically when the replication degree increases.

The rapid development of Group Communication Systems (GCS) [1] sheds a new light to this. Current research on data replication focuses on utilizing the rich multicast semantics provided by GCS to serialize transactions [17, 18, 19]. PostgreSQL-R [30] is a great success by introducing replication into PostgreSQL [25] using an eager update-everywhere strategy. It uses a particular multicast primitive provided by Spread [28], an advanced GCS, to propagate update information to all sites of

the system. This particular multicast primitive guarantees all running sites of the system deliver the same set of messages in the same order. That is, even if two sites multicast two different messages $m1$ and $m2$ simultaneously, all sites in the system receive them in the same order, i.e., either $m1$ before $m2$ or $m2$ before $m1$. In PostgreSQL-R, updates are multicast in a writeset message with this total order and then each site commits a transaction upon receiving its writeset. In such a way, all sites commit exactly the same set of transactions in exactly the same order, and concurrent transactions are serialized based on the order of writeset delivery. This keeps data consistent across the whole system without the need of distributed concurrency control or an agreement protocol. PostgreSQL-R has good availability and scalability without losing too much performance.

One very important issue which is not addressed sufficiently in the PostgreSQL-R project is *recovery*. Recovery, is the mechanism to bring a site, which either failed previously or is a brand new site to join the replicated database system, into the system.

Any replicated database system without a recovery mechanism is essentially impractical. First of all, any site is doomed to fail, in the long run. Since a replicated database system is designed to improve availability and reliability, any failed site must be able to rejoin the system. Second, the total workload submitted to the system usually increases over time. There must be a way to add new sites to the system to handle higher demands. The recovery mechanism is needed when a site joins or rejoins a running system. When a failed site wants to rejoin the system, it first needs to bring its local data back into a consistent state. In this local recovery, it makes

sure that all changes by transactions that committed before the crash at this site are reflected in its database, but none of the changes by aborted transactions. After that, it needs to update the local data to be consistent with other sites, which means that it needs to apply the changes of transactions that committed during its down time.

Recovery can be performed both off-line and on-line. Off-line recovery means that the system stops processing client requests during the time when a site is being brought into the system. The system only resumes its normal operations once recovery is done. On-line recovery, on the contrary, allows the system to keep on processing client requests even during the recovery of sites. For critical systems, like flight control system, only on-line recovery is practical. The challenge of on-line recovery is that recovery has to be synchronized with the execution of ongoing transactions at other sites. At the end, the recovering site should have the changes of these transactions. In this thesis, we focus on on-line recovery.

Recovery in a replicated database system can be performed using different strategies. We can either let a peer site transfer the information of transactions missed by the recovering site and re-execute these transactions at the recovering site, i.e., apply the changes of these transactions one after the other. Or we can take a snapshot of a peer site and install the snapshot at the recovering site. What are the significant factors that affect the performance of each strategy, and how to choose one over the other? These are issues we want to address in this thesis. In particular, this thesis presents an hybrid recovery mechanism that is able to perform both types of recovery and dynamically choose the one for which a faster recovery is expected. Furthermore,

the algorithm coordinates the recovery process with ongoing transaction processing at the other sites such that neither the peer nor the recovering site misses a transaction. We have implemented our approach into PostgreSQL-R and call the extended system PostgreSQL-RR

The rest of the thesis is structured as follows. Chapter 2 gives an introduction to group communication systems, the architecture of PostgreSQL-R and its concurrency and replica control algorithm. Furthermore, it gives an overview of different recovery strategies. Chapter 3 presents a formal description of a hybrid distributed recovery algorithm for PostgreSQL-RR. Chapter 4 shows how the algorithm is implemented in PostgreSQL-RR. Chapter 5 gives an evaluation of this implementation and compares different recovery strategies. Chapter 6 shows how the system handles network partition and total failure. Chapter 7 concludes the thesis.

CHAPTER 2
Background

This chapter goes over some prerequisites needed for the understanding of the PostgreSQL-RR system and the recovery algorithm implemented in the system. This includes an introduction to the main concepts behind group communication systems, an overview of the existing PostgreSQL-R system, and an outline of possible recovery strategies for replicated database management systems.

2.1 Group Communication Systems

More and more distributed systems are developed to solve more and more complex problems. An inherent difficulty in building a reliable and efficient distributed system is to determine the current state of remote components. Group communication systems (GCS) [1], which provide a messaging service and group membership service, have been developed to help solve this problem. Some well known GCS include ISIS [7], Transis [13], Totem [23], Horus [26] and Spread [28]. Spread is indeed used in PostgreSQL-R and PostgreSQL-RR because of its superior performance and functionality.

The membership service provides the group concept to let applications define multiple recipients of a multicast message. That is, *group* is an abstraction for a set of processes, and is identified by the group name. When the application needs to multicast a message to these processes, it just multicasts the message to the group. Another abstraction, *view*, is used by the GCS to keep track of the current group

composition. A view-change message is delivered to all members of the group should the group composition change. The GCS provides an interface to applications to define a group, to join or leave a group, and to get the up-to-date membership information of a group. The GCS also detects the failure of any group member and excludes it from the group.

The messaging service provides a rich set of multicast primitives to distributed applications. Multicast semantics can be categorized by two attributes: message ordering and message reliability. Messages can be ordered in one of the four ways: unordered, FIFO ordered, causally ordered and totally ordered. Only total order multicast is used in PostgreSQL-RR, hence is of our interest in the following discussions. In George Coulouris, Jean Dollimore, and Tim Kindberg's book [10], *total ordering* is defined as following: "If a correct process receives message m before it receives m', then any other correct process that receives m' will receive m before m'". Here, a *correct* process is a process that does not fail under the time of observation. So, total order multicast guarantees that all correct processes in the group receive messages in exactly the same order. The reliability attribute can take one of the following values: unreliable, reliable, and uniform reliable. Reliable delivery guarantees any message multicast by a correct process will be received by all correct sites eventually and at most once. This is sufficient only for systems that do not consider recovery. Consider the following scenario where a process multicasts a reliable message to the group, receives it and fails directly afterwards. For some reason, this message does not reach the other processes of the group and hence will not be received by them. This does not conflict with the reliable message property. However,

it has two negative implications. First, the application of the failed site might have performed some actions on behalf of the message before crashing (e.g., committing a transaction), but the other available sites are not aware of the message. Second, it poses a great challenge to the recovery of the failed process, because it received a message that no other process has received. Uniform reliable delivery addresses this issue and guarantees that if a process p receives a message m, then even if p fails directly afterwards, message m will be received by all other correct processes. This normally requires that if a message m is multicast to the group, the GCS component of each process first receives the message, then sends an acknowledgment to all others, and only when it receives acknowledgments from all other sites does it deliver the message to the application. That is, the application only receives a message when the GCS knows that every other process has physically received the message. This introduces a higher message delay compared to reliable delivery. Uniform reliable multicast is used in PostgreSQL-R.

2.2 Virtual Synchrony and Extended Virtual Synchrony

The Virtual Synchrony (VS) model is discussed in [6, 7, 15]. The VS model basically guarantees two properties. (1) View change messages, which are generated by the system when processes leave/join the group, are totally ordered. This means that all correct processes install the same set of views in the same order. (2) Other multicast messages generated by processes are totally ordered with respect to view change messages. This means that if a process delivers a message m in $view_i$, then no other process should deliver m in another view. If reliable multicast is chosen for

the VS model, we will have the following additional property: if V_1 and V_2 are consecutive views installed both at processes p and q, then p and q receive the same set of messages in V_1. Notice that the original VS model, which is implemented in ISIS [7], does not consider uniform reliable multicast, but paper [27] elaborates how the reliable multicast primitive can be extended to provide uniform reliability. Furthermore, in case of network partitions, VS only allows processes in the primary partition to progress. The primary partition is the partition that contains the majority of the group. Processes that are not in the primary partition should stop execution. They can later recover and rejoin the primary partition with a new identifier.

The Extended Virtual Synchrony (EVS) model is discussed extensively in paper [22]. EVS is defined by a set of specifications regarding how messages are delivered. The main difference to the VS model is specified by the *Safe Delivery* property, which requires a uniform reliable multicast. In fact, *safe* is just a synonym of uniform reliable total order delivery. Furthermore, EVS uses a different failure model than VS to allow network partitions and re-merges, and processes in the non-primary partition to continue to execute, yet the delivery of messages across the system remains consistent. EVS can be implemented on top of the message transmission, membership, and total ordering algorithms. The main idea is to switch GCS into a transitional phase before a new view is installed. GCS first delivers a transitional view change message to all group members to signal the start of the transitional phase, then it tries to recover lost messages and any safe messages that could not be delivered in the previous view. A lost message for site S_a is a message m multicast by any other site in the view but has not been received by S_a. So, in the transitional

phase, m must be retransmitted to S_a from another site in the transitional view. A safe message that could not be delivered is a message m that was received by S_a, but S_a has not yet received acknowledgment for the receipt of m from all other sites in the view. After all these messages are recovered and delivered to all connected members, the view change message is delivered and the new view is installed. Messages sent by members in the transitional phase are buffered, and they are delivered in the new view.

Spread [28] implements the EVS model and provides a uniform reliable total order multicast primitive.

2.3 PostgreSQL-R

This thesis enhances the PostgreSQL-R system to support recovery. Before introducing the enhancements, we first need to have some basic knowledge of the PostgreSQL-R system. PostgreSQL-R extends the open source database management system PostgreSQL with replication. The system and its data are fully replicated in a cluster of sites (i.e., computers). All sites of the cluster work collaboratively via uniform reliable and total order multicast messages to provide better system scalability and availability. The architecture of the system is discussed in Chapter 4, where we talk about how we integrate our recovery algorithm into the system.

2.3.1 Two running modes of PostgreSQL-R

PostgreSQL-R can run in two different modes: non-replication mode and replication mode. Running in the non-replication mode, a site takes client requests and

processes them locally. Modifications are only reflected in the local database. Running in the replication mode, the local system joins a communication group maintained by the GCS, and collaborates with other members of the group. Each site in the group can handle client requests separately, but any update made by a committed transaction to the local database is also propagated to all other sites and is applied to their copy of data. Accordingly, transactions are executed in different ways under these two modes.

Under the non-replication mode, any request submitted by a client is parsed, optimized and executed within the boundary of a transaction. Modifications made by a committed transaction are only reflected in the local database, so no writeset is created to capture the update information of the transaction. Upon the start of each transaction, a unique Local Transaction Identification (LID) is created and is assigned to the transaction. The system uses LID to identify transactions.

Under the replication mode, there's a global view and a local view of the PostgreSQL-R system. The global view of the system refers to the cluster of computers. Each of these computers runs an instance of PostgreSQL-R, and they communicate with each other through the GCS. The local view of the system refers the particular site of the cluster. When a transaction is submitted by a client to a local system, it is first executed at the local site and considered as a local transaction. A writeset structure (WS) is created for each local transaction to capture all update information of the transaction. Upon the client request to commit the transaction, the WS is propagated to all other sites via a total order multicast. When the WS arrives at a remote site, it is processed by this remote site in one transaction. This

transaction is considered as a remote transaction by this remote site. Conceptually, this remote transaction and the local transaction that originally created the WS should be considered as a single transaction, yet, they may have different LIDs at each site. PostgreSQL-R uses a unique Global Transaction ID (GID) to identify a transaction across the whole global system. It is very important to keep the GID for a transaction to be the same at all sites. Thanks to the total order multicast primitive provided by the GCS, this is relatively easy to accomplish by keeping a counter of all committed transactions at each replica. Upon the start up of the system, all counters start from one. Upon the delivery of a WS, the current reading of the counter is assigned to the transaction. If the transaction commits, the counter is increased by one, otherwise, the reading remains the same for the next transaction. Since all sites commit the same set of transactions in the same order, each committed transaction is assigned the same GID at all sites. Each replica also maintains a mapping between the LID and the GID of each committed transaction.

2.3.2 Concurrency and Replica Control for PostgreSQL-R

The challenge of replication is to detect conflicts between different transactions if the transactions execute at different sites. That is, if two transactions update the same data object at different sites, none of these sites is aware of the conflict occurring at the other site. Only when writesets are exchanged can such conflicts be detected and handled. In order to understand the concurrency and replica control mechanisms implemented in PostgreSQL-R, we have to understand how a central PostgreSQL sever performs concurrency control for local transactions running on a single site.

A database is a set of relations where each relation contains a set of records. A read operation returns some values from the current state of the database. A write operation modifies some values of the current state of the database and derives it to the next state upon the transaction commits. So, the history of the database is the ordered sequence of these states. A *database snapshot* [2] is an abstraction that captures one of these states of a specific moment of the evolution history of the database. *Snapshot Isolation* (SI) [5] is a type of multi-version concurrency control. Under SI, a transaction T_i is assigned the begin timestamp (either physical or logical) $TS_i(BOT)$ upon start. It then gets a snapshot of the current committed state of the database history. All subsequent read and write operations will be performed on this snapshot, hence are invisible to other transactions. At commit time, the transaction gets commit timestamp $TS_i(EOT)$. T_i conflicts with transaction T_j if $TS_j(EOT)$ in the interval of $[TS_i(BOT), TS_i(EOT)]$ (i.e., T_i and T_j are concurrent but T_j commits before T_i) and T_j wrote data that T_i also wrote, and we call this a write/write conflict. In this case, T_i must abort. Otherwise, it commits, and modifications to the snapshot will be reflected back to the database and derives it to a new state. Hence, state changes by a transaction are reflected in the global database state at the time of commit of the transaction. As you might notice, read operations do not block or conflict with other operations. This maximizes the potential for concurrent execution and boosts the system performance.

PostgreSQL uses a multi-version system to store its data [30]. Each update on a data record creates a new version. PostgreSQL keeps all versions of data (even the deleted records) in the system. Each data record is treated as an object. The

state of the database at time t is the collection that contains an appropriate version for each data object. The appropriate version for the data object X is the one created by the transaction T_j such that T_j is the last transaction that updated X and committed before t. This makes it relatively easy to take a snapshot of the database and implement an SI algorithm. When a data object X is first accessed by a transaction T_i, T_i retrieves the version from the state of the database at the time of the start of T_i, and copies it into its own snapshot. Subsequent modifications made by T_i are only reflected in this snapshot and are invisible to other transactions. When T_i commits, a new version of the modified data will be appended to the database, which will form the new state of the database. PostgreSQL uses two-phase-locking to serialize concurrent transactions that have write/write conflicts, which means that a transaction needs to acquire a lock on the data object before it writes it and releases the lock only after the transaction commits or aborts.

PostgreSQL-R [30] extends PostgreSQL's concurrency control algorithm to provide Snapshot Isolation (SI) to handle concurrent transactions in a replicated setting. The new algorithm is called SI-PR and describes how PostgreSQL-R handles local and remote transactions differently.

A local transaction is executed in three phases: execution phase, send phase and commit phase. During the execution phase, for each data object X accessed by transaction T_i, T_i retrieves the version V of X created by transaction T_j such that T_j is the last transaction that updated X and committed before T_i starts. If T_i contains only read operations on X, V is used. If T_i contains write operations on X, it first checks for write/write conflict. If there is a transaction that also wrote X and

committed after T_i started, T_i aborts. If there is no conflict, it requests a lock on X. If the lock is granted, it clones V to V', and performs subsequent updates on V' when the lock is granted. Otherwise, it waits for the lock. When the lock is eventually granted to T_i, T_i needs to reperform the conflict check. It also retrieves the GID of T_j from *pg transrecord*, and adds both V' and the retrieved global id GID to the writeset WS_i of T_i. Upon the commit request for T_i, if T_i is read-only, it commits right away and the send phase is skipped. Otherwise, the send phase begins. WS_i is sent to all replicas using total order multicast. The commit phase starts upon the delivery of WS_i. The GID_i is determined and is added to *pg transrecord* along with LID_i. Finally, T_i updates the log (to record the commit of the transaction), releases all locks it acquired in the execution phase, and wakes up all transactions that are waiting for any of these locks.

A remote transaction is executed in three phases: version check and early execution phase, late execution phase and commit phase. The version check and early execution phase starts upon the delivery of the WS_i for remote transaction T_i. GID_i is determined right away. A transaction T_i is then started with LID_i locally. For each object X in WS_i with associated GID_j, a version check is performed: T_i retrieves the local version V of X created by transaction T_k such that T_k is the latest committed transaction that updated X and committed before T_i starts. It then retrieves GID_k from *pg transrecord*. If GID_j is different from GID_k, which implies that another transaction at this remote site, T_k in this case, has updated X between the time T_i updated X on the local site and this version check, T_i must abort. T_k must be a transaction whose writeset was received between the time T_i's local site

multicast WS_i and the delivery of WS_i. As we said before, if two transactions that execute concurrently at different sites conflict, we commit the first whose writeset is delivered (in this case T_k) and abort the other (in this case T_i). If there is no conflict, T_i tries to acquire a lock on X. If the lock is granted, it updates the object and goes for the next object contained in WS_i. If the lock is being hold by a local transaction, that transaction is put into the *abortTransactionList*, and the update to the object X is delayed. If no other version check leads to a conflict, T_i may commit. As a result, all transactions in the *abortTransactionList* must abort. They are concurrent local transactions who have a conflict with T_i but their writesets have not yet been received (otherwise they would already have committed). T_i aborts all these transactions, and performs those postponed updates. If T_i aborts due to a conflict, none of these transactions will be aborted but they keep their locks. Furthermore, the GID_i that T_i receives will be reused by the next transaction whose writeset will be received. In the commit phase, T_i records the commit/abort of the transaction, releases all holding locks and wakes up transactions that are waiting for these locks.

2.4 Recovery in a Replicated Database Management System

Current research on recovery can be found in both the database systems community [20, 8] and the distributed systems community [24, 3] and cover both non-replicated and replicated systems. The basic ideas behind the recovery mechanisms proposed in theses communities are very similar. They all requires the system to do the following: 1) Regularly checkpoint the system, which means to log the current state of the system. 2) Log all changes in the order they were made to the system. 3) Upon the begin of the recovery, restore the system to the lastest checkpointed state,

and replay all changes logged after that checkpoint. For a non-replicated system, all logged information is usually kept in a local stable storage. For a replicated system, both the checkpointed state and the changes may be kept in the stable storage at a remote site, hence a *state transfer* may be required. As we can see later in this thesis, recovery in a replicated database management system follows the same path.

A replicated database management system is a system that replicates processes and data and distributes them into a set of connected sites, where each site runs more or less as an instance of the database management system. Recovery in such a system can be quite sophisicated, and is discussed in papers [20, 8]. Recovery in a replicated database system is required when a site, either a brand new one, or a site that failed previously, joins the system. The whole recovery process can be divided into two steps: *central recovery* and *distributed recovery.* Central recovery is only needed when the joining site was previously a member of the system. Central recovery brings the local database of the joining site back into a consistent state (i.e., modifications to the local database by all transactions that committed before the site's failure should be reflected in the database; modifications by any non-committed transaction must not be reflected in the database). Central recovery is provided by all database systems as a standard feature. It can be performed independently before distributed recovery. Distributed recovery makes the local database of the joining site to be consistent with other sites of the system. This subsection emphasizes on distributed recovery.

We are looking at two different strategies for distributed recovery: the Total Copy Strategy (TCS) and the Partial Copy Strategy (PCS). With TCS, a complete

copy of data stored in sites of the current system is first transferred to the joining site, then the joining site installs the copy into its local data storage. With PCS, only the update information of committed transactions that were missed by the joining site is transferred to the joining site, then the joining site installs the updates into its local data storage. There exist other variations of these strategies or different strategies. In case that a failed site rejoins the system, an enhancement can be made to TCS in that we only transfer the part of the database that has been modified by transactions since the previous failure of the joining site. That is, only the objects that were changed during the downtime are transferred. This is very attractive when many transactions were executed during the downtime of the recovering site, but they only changed a small active part on a huge database (i.e., hot-spot data). In this case, PCS would have to apply many transactions, although for each object only the last change is relevant. TCS would copy a lot of data that has actually not changed. However, determining what data objects actually changed is quite complicated and has its own overhead. There is also a *lazy* approach [16], where data objects are synchronized only when they were updated by a new committed transaction.

In theory, both TCS and PCS can be applied when distributed recovery is required. But in practice, both strategies have their advantages and limitations. When a new site joins a running system, PCS can not be applied unless the current system keeps a complete history of update information of transactions that have been committed since the original start of the system, which is very unlikely. Hence, only TCS is good for the task. When a site that failed previously is rejoining the system, both TCS and PCS can be used. TCS has the advantage that it does not require the

running system to maintain a log of the update information, while PCS requires the system to provide mechanisms to log and retrieve update information. The current system must have kept update information of all transactions missed by the rejoining site. The size of the updates may be huge for a high throughput and update intensive system or if the joining site has been down for a significantly long time before its rejoin. Even if all necessary updates are kept by the running system, the size could be so big that the recovery using PCS will be outperformed by a recovery using TCS in term of efficiency, and maintaining the update history may post significant overhead to the system. Mabrouk Chouk proposed in his master's thesis [8] a way to retrieve the update information from the Write Ahead Log (WAL), which is required by the central recovery anyways, to avoid keeping extra information. But this can only be a system specific solution. For instance, the WAL kept by the database may be checkpointed from time to time by the system automatically, and then the distributed recovery may fail. Even if we can have full control of the WAL, a further question then will be, how much update information should a running system keep, and how can it checkpoint its update information log. However, if the database is huge, the recovery time in case of TCS might be prohibitive and much longer than PCS, especially if the failed site was down for little time and has missed only few transactions.

Inherent challenges arise when TCS and PCS are performed online. Here, the rest of the system continue executing transactions. In this case, a synchronization point must be found. To find the synchronization point means to decide the transaction T such that missed transactions committed before T are transferred from a peer

site to the recovering site while T and all later transactions are applied at the recovering site as standard remote transactions. If TCS is chosen, the peer site providing the state might have to stop processing any new transaction during the recovery, or the state of the joining site after recovery will not be consistent with those of other sites of the system. This might dramatically reduce the availability of the system. Furthermore, the peer site needs to buffer all update information propagated from other sites during the recovery, which might not be feasible if the throughput of the current system is high and the recovery will take a long time.

At this point, you might want to ask, can we have a flexible solution, a hybrid one that combines the two strategies? Yes, we can! A distributed recovery algorithm that always chooses the optimal strategy, yet the system only keeps a reasonable size of update information, will be elaborated in Chapter 3.

CHAPTER 3
Distributed Recovery Algorithm for PostgreSQL-RR

This chapter describes our approach to distributed recovery at the algorithmic level, the implementation details of how it is integrated into PostgreSQL-RR will be elaborated in the next chapter.

3.1 Overview of the Algorithm

The distributed recovery algorithm used in PostgreSQL-RR is a hybrid one that combines both TCS and PCS. The algorithm uses a heuristic to choose the best strategy to perform the distributed recovery. In case on-line recovery is required, the recovery may need to use both strategies, one after another, to make the state of the joining site to be consistent with other sites of the running system. Before we formally present the algorithm in the next section, we would like to give an overview of the algorithm first.

In order to provide PCS recovery, the system needs to keep track of the updates of all transactions. As mentioned in the previous chapter, the WAL might be used but this is not possible in PostgreSQL-RR due to PostgreSQL's checkpointing mechanism. Instead, we changed the commit handling during usual processing. At the end of the successful execution of a transaction and just before the system can commit the transaction, the updates of the transaction, which are kept in the writeset data structure (WS) in PostgreSQL-RR, together with the GID of the transaction, are stored into a special table. We call this *Writeset Log* (WSL). This log can be

later used by the peer site to send the WSs of transactions the recovering site has missed. In order not to let WSL grow indefinitely, we still allow a site to checkpoint the log and delete the oldest update information.

The distributed recovery begins when a new site joins a running system or after a previously failed site restarted and finished the central recovery, and wants to join the system. This site S_j first joins the GCS group composed of sites of the running system. Then, S_j locates one of the group members S_p as the peer that will assist the distributed recovery process, and directly connects to it. S_j can locate potential peer sites by either looking at the membership of the current GCS or by being provided by a list of candidates via a configuration file specified, e.g., by the system administrator. In PostgreSQL-RR, the latter is used. S_j asks each potential candidate whether it is willing to become the peer. A site can deny the request because it is heavily loaded, for instance. After the connection is set up to a willing peer S_p, S_j tells S_p via the connection what transactions it has missed since its previous failure. As mentioned in the previous chapter, all committed transactions are totally ordered by the system, transactions are committed at all sites in exactly the same order, and each transaction is denoted by its unique Global Transaction Identifier (GID). In PostgreSQL-RR, GIDs are in consecutive, monotonically increasing order. So, S_j sends the GID (*lastGid*) of the last transaction committed at the site just before its previous failure. If S_j is a new site, it sends -1 as a predefined signal. When S_p receives *lastGid* from the connection, it estimates the cost of TCS recovery and the cost of PCS recovery and chooses the one with smaller estimated cost. S_p looks in its WSL for a record that contains *lastGid*. If the record can not be found, it means

S_p checkpointed its WSL, or S_p joined the system itself only after the failure of S_j, or S_j is just simply a new site being added into the system to share the workload. Hence, S_p does not have update information of all committed transactions that are missed by S_j. In this case, TCS has to be used. Otherwise, both TCS and PCS can be used. S_p retrieves the maximum GID *maxGid* contained in its WSL. The gap between *lastGid* and *maxGid* represents the number of committed transactions missed by S_j so far. If the gap is greater than a threshold, which is determined by a heuristic, it is assumed that a distributed recovery using TCS will outperform the one that uses PCS. Hence TCS will be used. If the gap is smaller than the threshold, PCS will be used.

If PCS is used, S_p retrieves all WSs whose GID are greater than *lastGid* from WSL and sends them to S_j. WSs are sent in multiple rounds. At each round, S_p packs a certain number of WSs into one message and sends it to S_j. Upon receipt of the message, S_j unpacks the WSs one by one from the message and applies them. Applying a WS is similar to executing a remote transaction, except no locks need to be held for data to be updated. After all WSs in the message have been processed, S_j notifies S_p to send more WSs. In offline recovery, the recovery is done when all retrieved WSs were sent to S_j and applied. In on-line recovery, new transactions are simultaneously executed in the running system. This means WSs of these newly committed transactions keep arriving at both S_p and S_j. Recall that S_j has joined the group and hence, receives all WSs currently multicast in the group. In this case, both S_p and S_j need to do more. S_j may receive the same WS both from S_p and the GCS. It needs to know when and how it should switch from applying WSs

received from S_p to applying WSs received from GCS. S_p needs to know up to which WS it should send to S_j, and must agree with S_j when to stop transferring WSs. A similar concept, called *determination of a synchronization point*, is mentioned in paper [20]. So, a synchronization protocol is needed here. In PostgreSQL-RR, when S_p perceives that there are only a few more WSs left to be sent to S_j, it multicasts a special message in total order to all sites. S_p then sends all WSs of transactions that commit before the receipt of this special message; and S_j starts to buffer WS received from GCS after the receipt of the special message. The switch will then be straightforward. When S_j has applied all WSs from S_p, it starts to apply the buffered WSs. When all buffered WSs have been applied, the recovery is done, and S_j can start to act as a normal site and to handle client requests.

If TCS is used, S_p first retrieves *maxGid* from WSL, and then takes a snapshot of the current state of its local database and saves it to a file. No transaction should be committed between the retrieval of *maxGid* and taking the snapshot. Taking a snapshot is an atomic operation and can be serialized with other transactions. So, the snapshot reflects all modifications made by transactions committed before taking the snapshot, and none of the modifications made by transactions committed after the snapshot. This property of snapshot is very important for the synchronization protocol. Furthermore, the maximum GID of transactions committed before the snapshot is *maxGid*. After the snapshot is taken, S_p sends *maxGid* to S_j. Upon the receipt of *maxGid*, S_j leaves the group and halts. The system administrator then copies the snapshot data from S_p to S_j, restarts S_j in non-replication mode, and installs the snapshot into its local database. After the snapshot is fully installed, S_j

contains data that is consistent with the data of S_p when the snapshot was taken. The system administrator restarts S_j again in replication mode. S_j joins the system, performs distributed recovery using PCS, with the *maxGid* received previously from S_p as its *lastGid*.

During the recovery, clients may not connect to the recovering site, since the transactions that handle these new client requests must be serialized after the transactions missed by the site during its down time.

Note however, that the peer site continues to execute local and remote transactions as any other non-recovering site during the recovery process.

3.2 Formal Description of the Algorithm

A recovery is a collaboration between the recovering site S_j and the peer site S_p, hence the recovery algorithm used in PostgreSQL-RR includes two parts: one for S_j and one for S_p. In the implementation chapter, we can see that both the peer and the recovery site are indeed composed of several processes. During the recovery process, these processes coordinate via messages. However, to make the algorithm easier to understand, it is described here at a relatively high level: each site is treated as a single unit, and communication between different processes within a site is hidden from the description. Furthermore, no further crashes are assumed during the recovery process.

3.2.1 Partial Copy Strategy

We present the recovery algorithm in two steps. First, we present the recovery algorithm that uses only PCS. Figure 3–1 shows the recovery steps at the recovering site S_j, and Figure 3–2 shows the recovery steps at peer site S_p. Then, we extend

the algorithm such that it can dynamically choose between TCS and PCS based on a cost estimation. Descriptions of constants and variables used in the algorithm can be found in Table 3–1, and Table 3–2 contains descriptions of some obvious functions in the algorithm.

Table 3–1: Constants and Variables

Vars/Constants	Description
N	The number of WSs that can be sent in one message.
$Nthreshold$	The criteria set by the user to be used in the heuristic to choose the recovery strategy.
$START\ BUFFER$	Boolean. S_j starts to buffer multicast WSs when it is set true.
$maxGid$	Maximum GID.
$type$	Message type. The message type determines the semantics of its content and how it can be decoded.
$NWS/AllWS$	WSs that are sent along a message.
$WSBuffer$	Linked list. Used by S_j to buffer multicast WSs.
$WSList$	Linked list. Used by S_p to store WSs retrieved from WSL.

As shown in Figure 3–1, the recovering site first performs local recovery, sets up some variables and then joins the replication group. As shown at line 7, we need to have a list of running sites from which the recovering site can choose the peer. There are different ways to have this potential peer list. One of them is to let the system administrator specify a list of potential peers in the configuration file. Or S_j can simply build a potential peers list from the view change message, which contains all connected members in the group. If the former is used and the system administrator can always determine the most appropriate site to be the peer, the *lastGid* can be sent along with the $MSG\ REQUEST$ message, and both the $MSG\ APPROVE$

Table 3–2: Functions

Functions	Description
$send(receiver, type, content)$	Sends a message.
$recv(sender, type, content)$	Receives a message.
$mcast(type, content)$	Multicasts a message.
$mcast\ recv(type, content)$	Receives a multicast message.
$buildWSList(fromGid, toGid)$	Retrieves WSs from WSL and inserts them into $WSList$. Details can be found in Figure 3–2.
$sendWS()$	Removes WSs from WSList and sends them to the recovering site. Details can be found in Figure 3–2.
$applyWS(WS)$	Performs the updates captured in the WS.
$connect(site)$	Builds a TCP/IP connection with the *site*.
$isToAssistRecovery()$	Used by a site to determine if it wants to be a peer site. The decision is based on its current workload and if it has been already assisting another site to recover. Details can be found in Figure 3–2.
$endAssistance()$	Called by a peer site when the recovery is done to do some necessary cleanup so that it can potentially assist another site to recover.

1. S_j starts up
2. Performs Central Recovery
3. $START\ BUFFER := false$
4. Joins the communication group
5. Retrieves $maxGid$ from WSL
6. $lastGid = maxGid$
7. For each potential peer P {
8. $connect(P)$
9. $send(P, MSG\ REQUEST)$
10. $recv(P, type)$
11. IF $type = MSG\ APPROVE$
12. $S_p := P$;
13. $send(S_p, MSG\ LAST\ TXN, lastGid)$
14. BREAK;
15. ELSE
16. // $type = MSG\ DENY$
17. IF all potential peers have been tried
18. // Recovery failed
19. System exits
20. }
21. FOR (; ;) {
22. Upon receipt of a connection request from a client
23. Declines the request;
24. Upon receipt multicast $(MSG\ WRITESET, WS)$
25. IF $START\ BUFFER = false$
26. Drops WS
27. ELSE
28. Appends WS to $WSBuffer$
29. Upon receipt $(S_p, MSG\ TXN\ UPDATE, NWS)$
30. For each WS in NWS
31. $apply(WS)$
32. $send(S_p, MSG\ CONTINUE)$
33. Upon receipt multicast $(MSG\ SYNC)$
34. IF the message was multicast by S_p
35. $START\ BUFFER := true$
36. Upon receipt $(S_p, MSG\ RECOVERY\ DONE, AllWS)$
37. For each WS in $AllWS$
38. $applyWS(WS)$
39. Blocks GCS channel
40. For each WS in $WSBuffer$
41. $applyWS(WS)$
42. Unblocks GCS channel
43. // Recovery is done
44. Runs as a normal site;
45. }

Figure 3–1: PCS Recovery at the Recovering Site

```
1. new WSList()
2. FOR (; ;) {
3.   Upon receipt (P, MSG REQUEST)
4.     IF isToAssistRecovery() = true
5.       S_j := P
6.       send(S_j, MSG APPROVE)
7.     ELSE
8.       send(P, MSG DENY)
9.   Upon receipt (S_j, MSG LAST TXN, gid)
10.    // Uses PCS
11.    buildWSList(gid, -1)
12.    sendWS()
13.  Upon receipt (S_j, MSG CONTINUE)
14.    IF WSList.size <= N
15.      buildWSList(0, -1);
16.    sendWS()
17. }

18. isToAssistRecovery() {
19.   IF assisting another site for recovery
20.     RETURN false
21.   ELIF system is heavily loaded
22.     RETURN false
23.   ELSE
24.     RETURN true
25. }

26. buildWS(fromGid, toGid) {
27.   IF fromGid = 0
28.     maxGid := max GID in WSList
29.   ELSE
30.     maxGid := fromGid
31.   IF toGid = -1
32.     Retrieves WSs whose GID are greater
      than maxGid and inserts them to the end
      of WSList
33.   ELSE
34.     Retrieves WSs whose GID are greater
      than maxGid but not greater than toGid
      and inserts them to the end of WSList
35. }

36. sendWS() {
37.   IF WSList.size > N
38.     Removes the first N WS from WSList
      and includes them in NWS
39.     send(S_j, MSG TXN UPDATE, NWS)
40.   ELSE
41.     mcast(MSG SYNC)
42.     mcast recv(MSG SYNC)
43.     upToGid := GetCurrGID()
44.     buildWSList(0, upToGid)
45.     Removes all WS from WSList
      and puts them in the AllWS
46.     send(S_j, MSG RECOVERY DONE, AllWS)
47.     endAssistance()
48. }
```

Figure 3–2: PCS Recovery at the Peer Site

message and the MSG LAST TXN are not needed. In PostgreSQL-RR, the former way is implemented (line 11-19).

During recovery, clients may not connect to S_j (line 22-23). WSs that are received from the GCS are either ignored or buffered (line 24-28). Only those WSs transferred from S_p are processed by S_j (line 29-31). When S_j receives the $MSG\ SYNC$ message, it knows that recovery is nealy done and it has to start buffering WS coming from the GCS (line 33-35). After S_j receives the $MSG\ RECOVERY\ DONE$ message from S_p and processes all WS that are included in the message (line 36-38), a transitional phase starts (line 39-42). S_j furst blocks the communication channel with GCS, that is, does not listen to the messages from GCS. It then processes all WSs that were buffered during the recovery phase, and at last reopens the communication channel again. So, during this transitional phase, WS that are multicast by other sites are buffered in the GCS. The reason that we have two buffers to store WS is because GCS and the PostgreSQL-RR are loosely coupled. If we allow new WS to be added to $WSBuffer$ during the transitional phase, we might remain there forever and won't switch back to normal mode. Instead, we want to make the transitional phase as short as possible. For PostgreSQL-RR, in all tests conducted, only one WS is buffered in $WSBuffer$ and needs to be processed if no more than 10 WS are sent in one single message from S_p.

Figure 3–2 shows the actions at the peer site. After agreeing to help with recovery (line 3-8 and 18-25), it sends writesets in several rounds (line 9-17). It continuously retrieves writesets from WSL (line 26-35) and sends them to S_j until the synchronization point is found (line 36-48).

3.2.2 Total Copy Strategy

As discussed before, TCS is necessary in some cases: (1) A new site is added to the cluster. (2) A joining site has crashed for a sufficiently long time before its restart and all other sites of the system checkpointed their WSLs in between. Furthermore, if the size of the database is relatively small, and the recovering site missed a lot of update transactions, a TCS recovery is preferred.

When TCS is performed, recovery is done in two steps. First, the peer site takes a snapshot of its database. For that it starts a transaction T which then reads the entire database. Then this snapshot is transferred to the recovering site and installed there. Note that the snapshot contains effects of all transactions committed before T started and none afterwards. Thus, all transactions concurrent to T will their changes not have reflected in the snapshot. Hence, the TCS will be followed by a PCS to retransmit these missing transactions. We can easily extend the algorithms displayed in the previous figures.

For the peer site, we can insert the code displayed in Figure 3–3 into Figure 3–2 between line 9 and 10. The peer site first decides whether to use PCS or TCS (line 2-5). If the recovering site has missed more than *Nthreshold* update transactions, TCS will be used. TCS must also be used if S_j is a new site. Recall that when a new site joins the system, its WSL is empty. When such site retrieves *maxGid* from its WSL (in Figure 3–1 at line 5), *maxGid* is assigned -1. Then, the recovering site sends -1 as the *lastGid* in $MSG\ LAST\ TXN$ message. When the peer site receives it and tries to find it from its WSL (in Figure 3–3 at line 5), it cannot be found. Hence TCS will be used. If TCS is used, S_p takes a snapshot as described above and

1. //Upon receipt ($S_j, MSG\ LAST\ TXN, gid$)
2. //Determines which recovery strategy to use
3. $maxGid$ = the max GID in WSL
4. $diff = maxGid - gid$
5. IF (gid cannot be found in WSL OR $diff > Nthreshold$)
6. // Uses TCS
7. $toGid := GetCurrGID()$
8. $send(S_j, MSG\ USE\ TCS, toGid)$
9. Takes a snapshot of database and saves it to a file
10. $endAssistance()$;
11. // Uses PCS

Figure 3–3: Extension for the Peer Site

1. Upon receipt ($S_p, MSG\ USE\ TCS, lastGid$)
2. Writes $lastGid$ to a file
3. Prints to screen to notify the system administrator about using TCS
4. System exits

Figure 3–4: Extension 1 for the Recovering Site

informs S_j about the last transaction included in this snapshot. Then the recovery is finished for the peer (line 6-10).

For the recovering site, the code displayed in Figure 3–4 needs to be added to Figure 3–1 between line 28 and 29. The basic idea is that when the recovering site receives the $MSG\ USE\ TCS$ message, it stores the gid in a file, signals on the screen that TCS needs to be used, and then shuts itself down. The system administrator's help is now needed. When the system administrator gets the signal, he waits until the snapshot is taken at the peer site. He then copies the snapshot from the peer site to the recovering site, restarts S_j in non-replication mode and installs the snapshot. After the snapshot has been fully installed, he shuts down S_j and restarts it at replication mode again. When now S_j again restarts, it must know whether a snapshot was installed. For that we have to also replace code displayed in Figure 3–1 at line 5 and 6 with code displayed in Figure 3–5.

1. IF *lastGid* is specified in the configuration file
2. Retrieves *lastGid* from the file
3. ELSE
4. Retrieves *maxGid* from WSL
5. *lastGid* := *maxGid*

Figure 3–5: Extension 2 for the Recovering Site

3.2.3 Discussion

I want to discuss a little bit more about some constants used in the above algorithms.

First, N, the number of WSs that should be sent in one single message, is determined by the buffer size of the Internet socket used for the communication between the peer site and the recovering site. The larger the N is, the less message rounds are required, but a larger buffer is required for the Internet socket. A large N also has a side-effect on the transitional phase. Since more WSs need to be sent in one message, it takes longer for S_p to copy the serialized WSs into the message and the message to be transferred from S_p to S_j. This implies that there will be a longer time after S_j sends $MSG\ CONTINUE$ and before it receives the next set of missed WSs. Hence more WSs may be buffered in $WSBuffer$ at S_j during the recovery phase. This may prolong the transitional phase. According to my tests, the benefit gained from using large N is not significant. A 10 for N is recommended.

Second, the value of *Nthreshold* represents the criteria used by the user to choose the recovery strategy, and it varies from case to case. Some major factors that need to be considered include the size of the database, the number of WSs that need to be transferred, and the average size of these WSs. Some guidelines based on test results will be given in Chapter 5.

Although we did not consider further crashes during the recovery, we can easily extend the algorithm to handle them. If the peer site fails at any time during the recovery, the recovering site terminates the recovery and exits. This will require the system administrator to restart the site again. The WSs that have already be transferred to and been processed by the recovering site do not need be redone during the next round of recovery. Similarly, if the recovering site crashes during the recovery, the system administrator just starts the site again.

CHAPTER 4
Integration of Recovery into PostgreSQL-RR

This chapter discusses how the recovery algorithm described in Chapter 3 is implemented in PostgreSQL-RR. Some important issues that are covered in this chapter include the architecture of PostgreSQL-R and how it is extended to PostgreSQL-RR, the role each component of the system plays during recovery and the interactions between components. Furthermore, the implementation of the WSL and how a snapshot is taken and installed are discussed.

4.1 Architecture of PostgreSQL-R

PostgreSQL-R itself is an extension of the open-source database management system PostgreSQL. As depicted in Figure 4–1, a PostgreSQL-R system is a cluster of nodes, and each node contains a full copy of data and is composed of the following components: *postmaster*, *replication manager*, *local backend*, *replication backend* and *communication manager*. The group communication system, in this case Spread, is not part of the system, but is required to work with the system to provide membership service and the uniform reliable and totally ordered multicast service.

Each component of the system is indeed a process. Postmaster is the main process that creates all other processes of the system. It first creates the communication manager and the replication manager upon startup of the system, and then creates the remote backend upon the replication manager's request. Upon a connection request from a client, it creates a local backend. A local backend is a process that

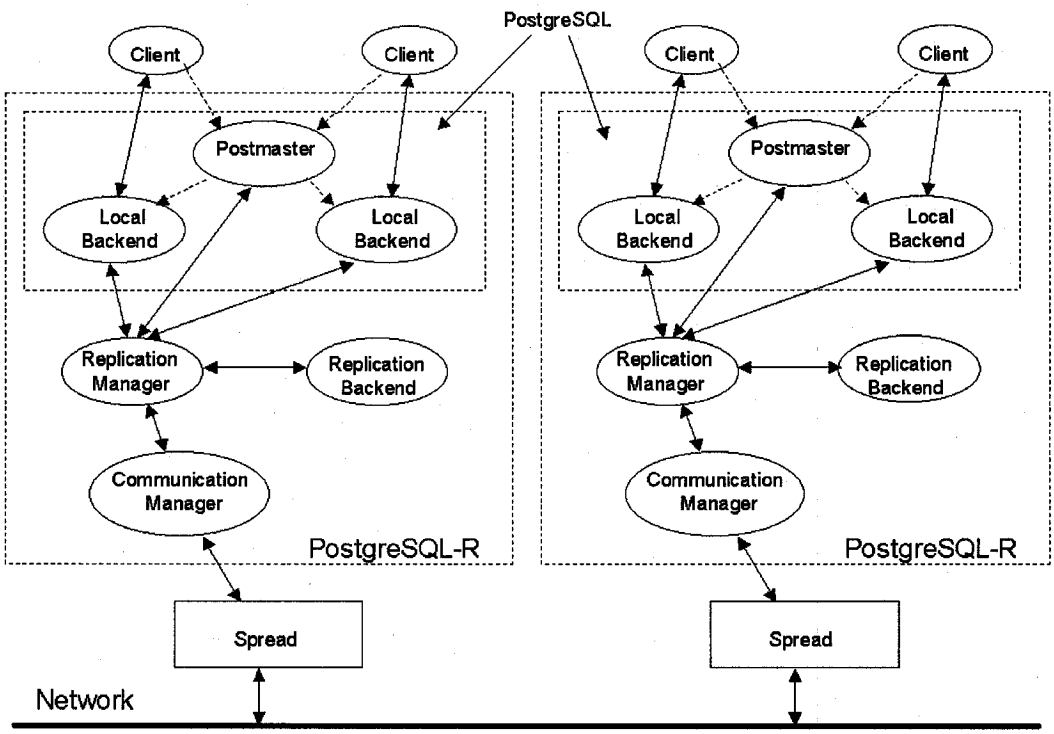

Figure 4–1: Architecture of PostgreSQL-R

takes client requests in the form of SQL statements and executes them locally. A replication backend processes writesets (WS) propagated from other nodes of the system. The replication manager coordinates the work of other processes and the communication between them. The communication manager provides an abstraction of the GCS to the replication manager.

4.2 Architecture of PostgreSQL-RR

The architecture of the new PostgreSQL-RR is very similar to PostgreSQL-R, as what can been seen in Figure 4–2. PostgreSQL-RR processes transactions in almost the same way, except some extra information needs to be logged for assistance of

recovery. That is, the new system basically inherits all components from PostgreSQL-R. Most components assume the same old responsibilities they do in the old system. A new component, the recovery backend, is added into the system, to retrieve WSs from the WSL at the peer site and transfer them to the recovering site. Then, the recovery backend at the recovering site processes the WS to update its local database.

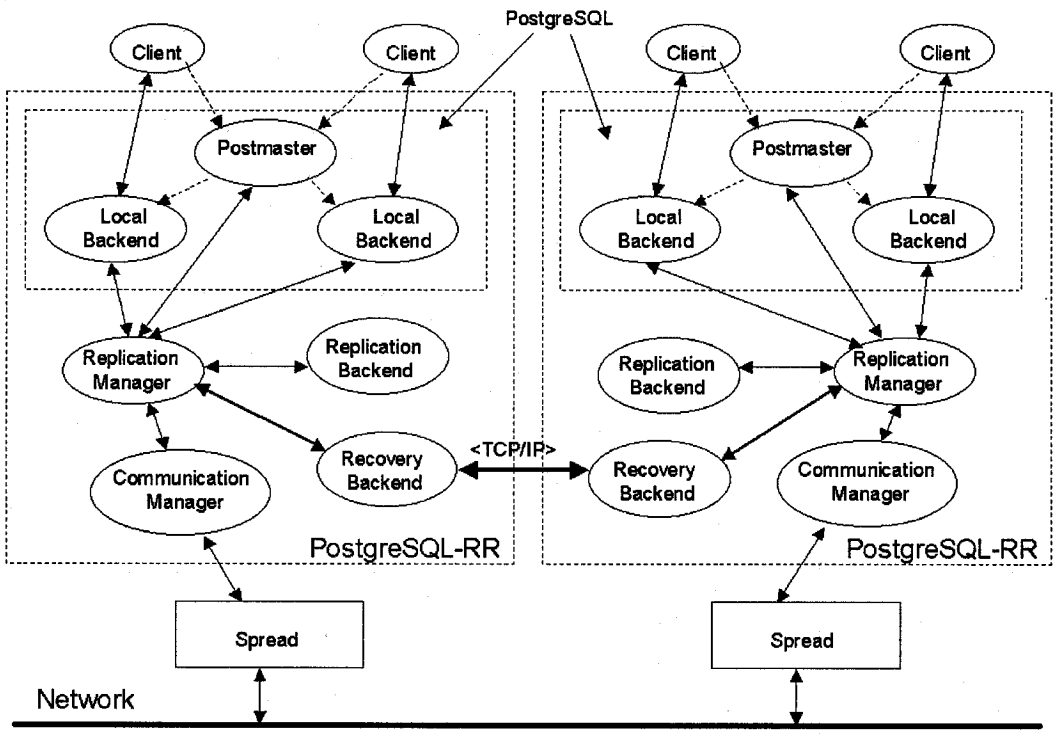

Figure 4–2: Architecture of PostgreSQL-RR

4.3 Major Players during Recovery

Recovery backend, replication manager, replication backend, and communication manager work together for recovery. The communication manager still works as a delegate for the GCS in the new system as it does in PostgreSQL-R, and the

replication backend at the recovering site needs to process all buffered WSs before the joining site can switch to a regular mode, but it processes these WSs the same way it processes remote transactions in the old system. Hence there was no need to change them in order to introduce recovery. So, this section focuses on the recovery backend and the replication manager.

Processes in the system are coordinated via messages, hence each of them is designed as a message driven system. The best formalism to describe a message driven system is UML *StateMachine* (SM). We give a brief introduction to this formalism, more details can be found in [29]. In a SM diagram, rounded rectangles represent state and arrows represent transitions. A *state* can be either a simple state or a composite state, which has nested internal states and transitions. Each *transition* may have an event[condition]/action rule where each element is optional. A transition is a system progression from its current state pointed by the arrow tail to its destination state pointed by the arrow head. A transition is fired when the *event* specified for the transition occurs and the *condition* is true, the *action* is executed before the system enters the destination state. The system starts from the *initial state* which is represented by the closed circle, and ends at the *final state* represented by the bordered circle. A circled X represents a *port*, which is the communication interface between the internal and the external of a state.

4.3.1 Recovery Backend

The recovery backend plays a major role in recovery. The recovery backends of the peer and the recovering site have different tasks. At the peer site, it retrieves

WSs from the WSL and transfers them to the recovering site. Then the recovery backend at the recovering site processes the WSs and updates the local database.

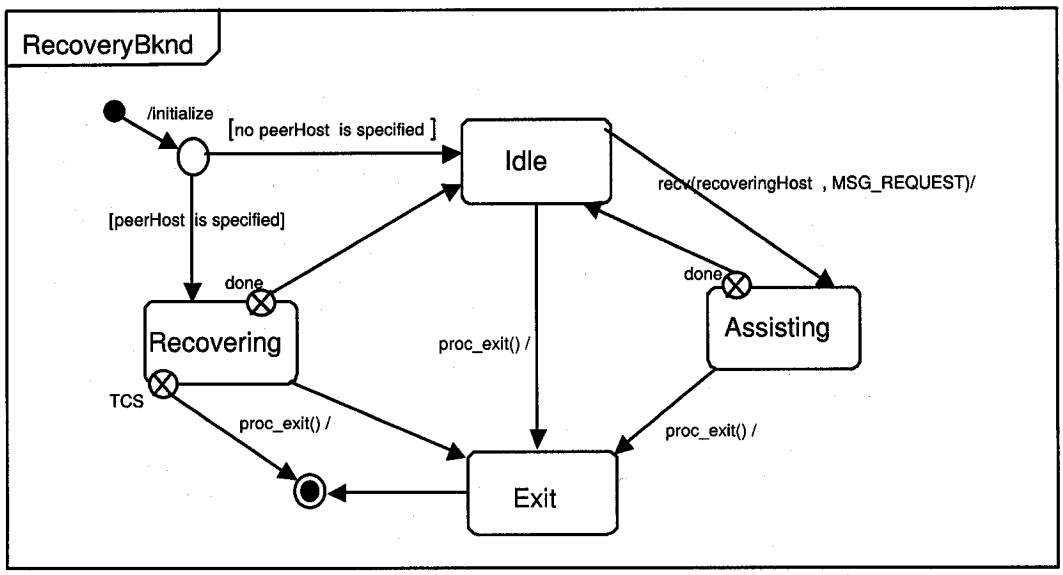

Figure 4–3: StateMachine for Recovery Backend

The SM in Figure 4–3 describes the dynamic behavior of the recovery backend, and the description of each message used in the recovery can be found in Table 4–1. The recovery backend can be in any of the following states: *idle*, *recovering*, *assisting*, and the *exit* state. Upon start up, the recovery backend initializes some data structures and the communication channel with the replication manager. After that, as mentioned in the previous chapter, it needs to get the potential peer list from which it can get a peer to assist its recovery. In PostgreSQL-RR, this list is given by the system administrator in the configuration file. For that, it checks in the configuration file if a peer host name is specified. If no, it enters the *idle* state, and no recovery

Table 4–1: Recovery Messages

Message Type	Description
MSG ASSIST	Sent by the recovey backend at the recovering site to the recovery backend at the peer site.
MSG APPROVE	Sent by the recovery backend at the peer site to the recovery backend at the recovering site.
MSG DENY	Sent by the recovery backend at the peer site to the recovery backend at the recovering site.
MSG LAST TXN	Sent by the recovery backend at the recovering site to the recovery backend at the peer site.
MSG TXN UPDATE	Sent by the recovery backend at the peer site to the recovery backend at the recovering site. The message contains up to N WS information.
MSG CONTINUE	Sent by the recovery backend at the recovering site to the recovery backend at the peer site.
MSG SYNC	It is sent by the recovery backend at the peer site to the replication manager at the peer site. The replication manager forwards it to GCS.
MSG SYNC GID	Sent by the replication manager at the peer site to the recovery backend at the same site.
MSG RECOVERY DONE	If it is sent by the recovery backend at the peer site to the recovery backend at the recovering site, it means the peer has transferred all WS needed by the recovering site. If it is sent by the recovery backend at the recovering site to the replication manager at the same site, it notifies the replication manager to start processing the buffered WSs.
MSG USE TCS	Sent by the recovery backend at the peer site to the recovery backend at the recovering site. It tells the recovering site that TCS is chosen to perform the recovery. The snapshot of the database reflects all transactions that have GID not greater than *lastGid*.

needs to be done for the site. This is used when we initially start up the cluster where all sites have the same database state. Otherwise, it enters the *recovering* state, and the site becomes a recovering site. If the recovery backend exits the *recovering* state via the *TCS* port, it enters the *exit* state, because if TCS is performed, the site has not yet completed recovery. Instead, it has to first apply the snapshot and then perform PCS. For PCS, after the recovery is done, the recovery backend enters the *idle* state. In the *idle* state, upon the receipt of a *MSG REQUEST* message from another recovering site, the recovery backend enters the *assisting* state, and the site becomes a peer to assist the recovery. When the recovery is done, or the recovering site failed during the recovery, the recovery backend at the peer site goes back to the *idle* state. During the recovery, the recovery backend at the recovering site is in the *recovering* state, while the recovery backend at the peer site is in the *assisting* state. If any error occurs at the site and function *proc exit*() is called, the recovery backend enters the *exit* state and does some necessary clean-up. After that, the process exits.

As illustrated in Figure 4–4, the *recovering* state is a composite state. Upon entering the state, the recovery backend first sets up a TCP/IP socket with the recovery backend at the peer site. After the communication channel is up, it sends a *MSG REQUEST* to the peer via the channel and enters the *waiting* state to wait for the response from the peer. In the waiting state, upon the receipt of message *MSG DENY*, it connects to another potential peer and requests assistance. In this case, the recovery backend remains at the same state. Upon the receipt of message *MSG APPROVE*, the recovery backend retrieves from WSL *lastGid*, the GID of the last transaction committed before the previous failure of the site. Note that if the

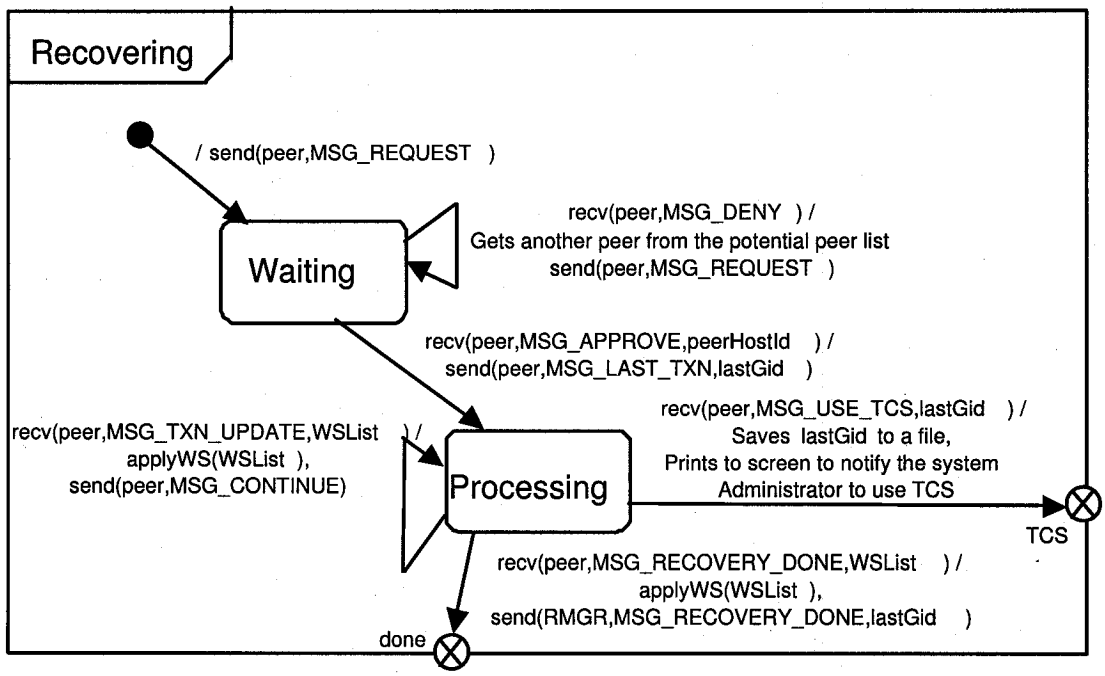

Figure 4–4: Composite State: Recovering

site is a new site to join the system, the WSL is empty, and the *lastGid* is set to -1. It then sends *lastGid* along with message $MSG\ LAST\ TXN$ to the peer and enters the *processing* state. In this state, the recovery backend responds to three types of messages. Upon receipt of message $MSG\ USE\ TCS$, it retrieves the *lastGid* from the message and writes it to a file. Then, it prints to the screen to notify the system administrator to assist the TCS recovery. After that, it exits the state via the TCS port. Upon receipt of message $MSG\ TXN\ UPDATE$, it retrieves the list of WSs from the message, and applies them one by one to its local database. Furthermore, it

puts the WSs into its local WSL. The WSs that come along with the message are already serialized and can be put into the WSL right away without further format conversion. Upon receipt of message *MSG RECOVERY DONE*, which basically tells the recovering site that all WSs missed by the recovering site have been transferred from the peer, the recovering site processes the received WSs, retrieves the maximum GID from WSL and sends it along with message *MSG RECOVERY DONE* to the replication manager. At this point, the recovery backend has just finished its part of the recovery, and enters the idle state. The recovery of the site is not done yet. As we will see later, the site still needs to transit to the regular state, which is performed by the replication manager in the *post-recovering* state (i.e., the transitional phase at the recovering site). The SM of the replication manager is displayed in Figure 4–6.

Similarly, the *assisting* state is also a composite state, which is illustrated in Figure 4–5. Upon entering the *assisting* state, the recovery backend is first in the *waiting* state. Upon receipt of message *MSG LAST TXN*, it retrieves *lastGid* from the message and uses it to determine which recovery strategy to be used, i.e., TCS or PCS. The decision is based on the estimation of the recovery time for using each strategy. If TCS is chosen, the recovery backend enters the *dumping* state. In this state, the recovery backend sends the maximum GID in the WSL to the recovering site in message MSG USE TCS. Then, it invokes the Unix system command to start a new shell to execute *pgdump*, a built-in feature of PostgreSQL, to take the snapshot of the database in one transaction. Note that no transaction should be committed during the sending of the message and the opening of the snapshot transaction, or it

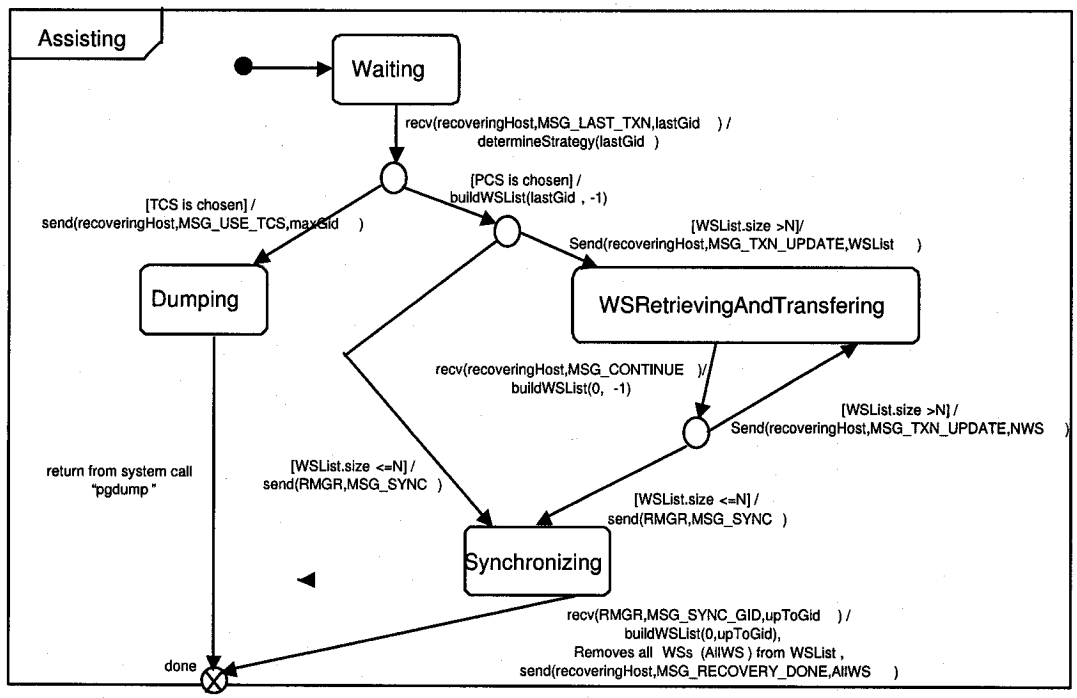

Figure 4–5: Composite State: Assisting

will be missed by the recovering site. When *pgdump* is done, the recovery backend exits the assisting state and enters the idle state. If PCS is chosen, the recovery backend retrieves all WSs whose GID is greater than *lastGid* into the $WSList$. If the $WSList$ contains no more than N ($N = 10$ in our implementation) WSs, it sends message $MSG\ SYNC$ to the replication manager and enters the *synchronizing* state. Otherwise, it removes the first N WSs from the $WSList$, sends them to the recovering site in message MSG TXN UPDATE and enters the *WSRetrievingAndTranserfering* state. While in the *WSRetrievingAndTranserfering* state, upon receipt of message $MSG\ CONTINUE$ from the peer, the recovery backend checks the number of WSs

contained in the *WSList*. If the number is greater N, it removes the first N WS from the *WSList* and sends them in the message *MSG TXN UPDATE* to the recovering site. If the number is no greater than N, it first retrieves all new WSs from the WSL (writesets that were entered into WSL since the last time the peer site retrieved from WSL). If the list size is larger than N, it sends again the first N WSs in the *MSG TXN UPDATE* message. Otherwise, it sends message MSG SYNC to the replication manager and enters the *synchronizing* state. In the *synchronizing* state, upon receipt of message *MSG SYNC GID* from the replication manager, it gets *upToGid* from the message, retrieves from WSL all WSs that have not been retrieved and have GID less than or equal to *upToGid* into the *WSList*. After that, it removes all WSs from the *WSList* and sends them in message *MSG RECOVERY DONE* to the recovering site. After this message is sent, the assistance of the recovery is finished, and the recovery backend exits the *assisting* state via the *done* port and enters the *idle* state.

4.3.2 Replication Manager

Apart from taking charge of information flow between components, as it does in PostgreSQL-R, the replication manager also helps in the synchronization phase of a recovery.

As depicted in Figure 4–6, the replication manager can be in any of the following states: *regular*, *recovering*, *transition* and the *exit* state. Upon start up, the replication manager checks in the configuration file if any potential peer site is specified. If the system administrator did not specify any in the configuration file, which typically tells the system no recovery is needed, the replication manager enters the *regular*

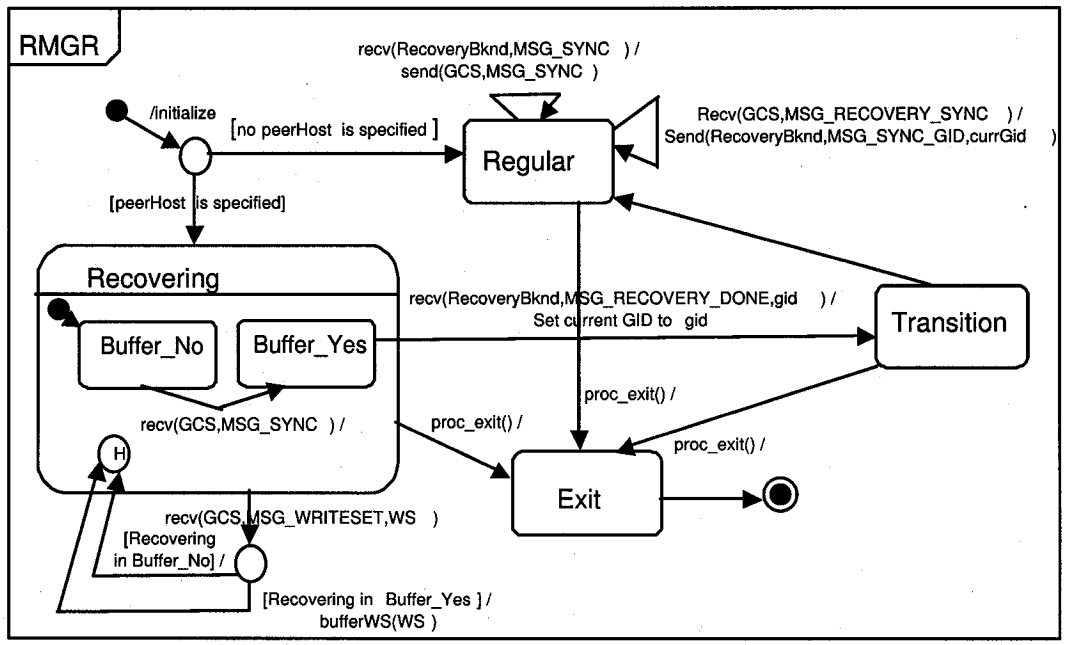

Figure 4–6: StateMachine of Replication Manager

state. Otherwise, the replication manager enters the *recovering* state. In the *regular* state, the replication manager behaves almost the same as it does in PostgreSQL-R. The only difference is that in PostgresSQL-RR, it needs to assist the synchronization of WS. Upon receipt of message $MSG\ RECOVERY\ SYNC$ from the recovery backend when the site is a peer assisting a recovering site for the recovery, it forwards the message to GCS, and GCS will then multicast the message in total order. Upon receipt of this message back from GCS, the replication manager sends the current GID to the recovery backend in MSG SYNC GID message. The *recovering* site is a composite state. Before the synchronization is done, the replication manager stays in the *Buffer No* state. While in this state, it drops all WSs received from GCS. Upon

receipt of message *MSG RECOVERY SYNC* from the peer site through the GCS, it enters the *Buffer Yes* state and starts to buffer every WS received from GCS. Upon receipt of message MSG RECOVERY DONE from the recovery backend, it exits the *recovering* state and enters the post-recovering state. In this state, it first blocks the communication channel with GCS, then sends all buffered WSs to the replication backend one by one. The replication backend processes them the same way it processes any other remote transaction. After all buffered WSs are processed, the replication manager reopens the communication channel with GCS and enters the *regular state*. The recovery is done. Again, the replication manager exits upon the invocation of function *proc exit*().

4.3.3 PostgreSQL-RR in Recovery

In this subsection, we assemble all pieces together to have a whole picture of the recovery. A recovery scenario where the recovery is done successfully using PCS is illustrated in Figure 4–7.

The recovery process is a sequence of interactions between processes at both the recovering site and the peer site. In the following discussion, we focus on the messages sent back and forth. Directly after the recovering site starts up, joins the GCS group and connects to the peer, the recovery backend of the site (*rcbr*) sends a *MSG REQUEST* message to its counterpart of the peer site (*rcbp*). *Rcbp* responds with the approval message *MSG APPROVE*. Then, *rcbr* retrieves the maximum GID from its WSL and sends it to *rcbp* in the *MSG LAST TXN* message. Upon receipt of the message, *rcbp* calls *buildWSList*() to retrieve all WSs whose GID is greater than the received *lastGid* and appends then to the end of the linked list

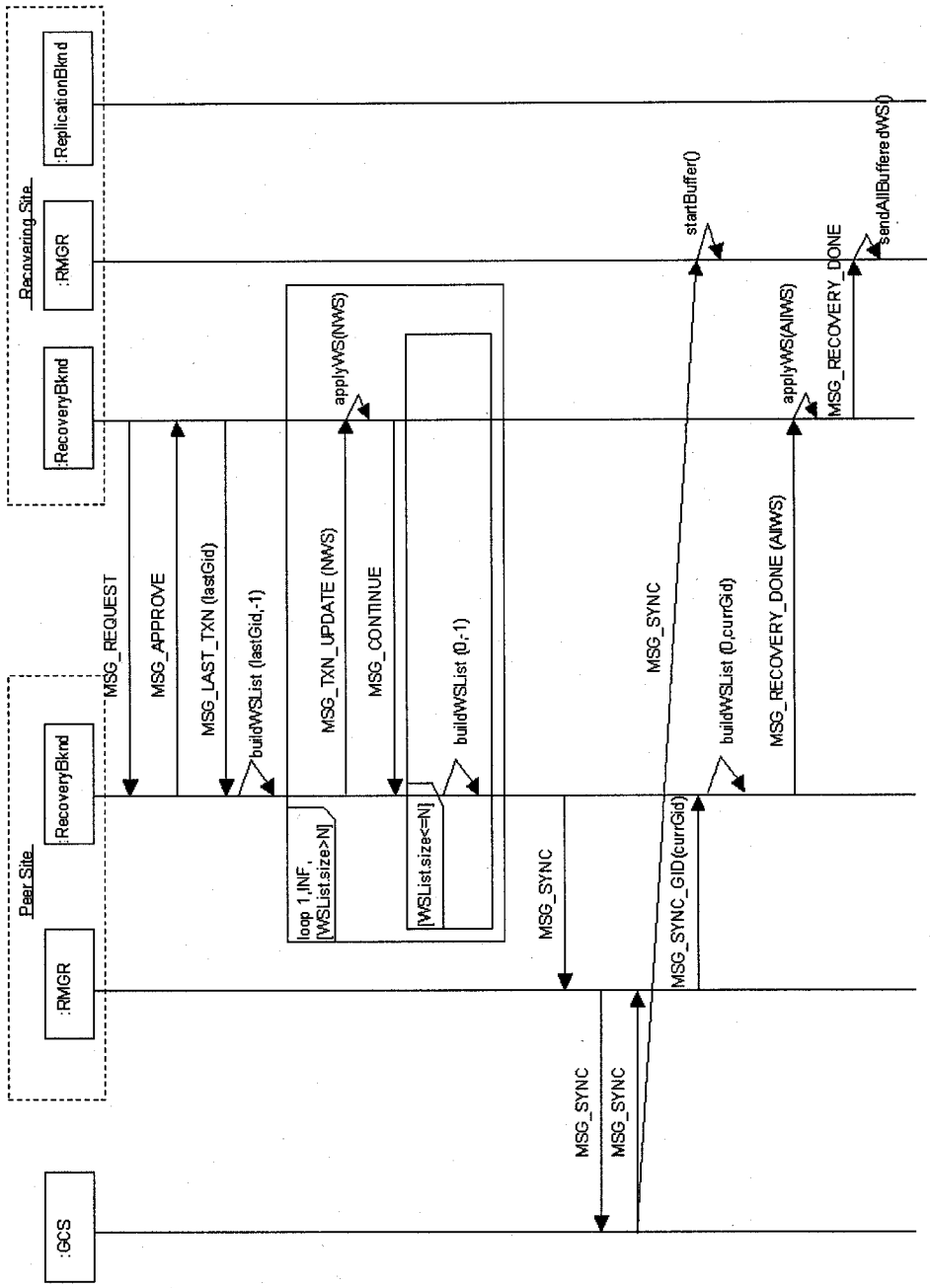

Figure 4–7: Recovery using PCS

WSList. Then, it starts a loop to send these WSs to *rcbr*. Each round, *rcbp* removes N WSs from the beginning of the *WSList* and sends them to *rcbr* in message *MSG TXN UPDATE*. Upon receipt of this message, *rcbr* extracts the WSs from the message and invokes *applyWS*(). For each WS, *applyWS*() basically opens a new transaction to apply the updates captured in the WS to its local database. The WS as well as its GID is also logged into the WSL. When all received WSs have been processed, *rcbr* sends message *MSG CONTINUE* to *rcbp*. Upon receipt of the message, *rcbp* checks if the *WSList* contains more than N WSs. If yes, another around of WS transfer begins. If the *WSList* contains no more than N WSs, *rcbp* invokes *buildWSList*() and then checks the condition again. If the list size is greater than N, a new round of WS transfer begins. If the list size is still not larger than N, the synchronization phase begins. For that, *rcbp* sends message *MSG SYNC* to the replication manger of the peer site (*rmgrp*). *Rmgrp* then forwards the message to GCS. GCS then multicasts the message. At the recovering site, upon receipt of the multicast message, the replication manager (*Rmgrr*) calls *startBuffer*() to set variable *START BUFFER* to *true*. From then on, and up to the site is fully recovered, *Rmgrr* appends WSs received from GCS in message *MSG WRITESET* into linked list *WSBuffer*. At the peer site, upon receipt of message *MSG SYNC*, *rmgrp* get the maximum GID (*currGid*) from its WSL and sends it to *rcbp* in message *MSG SYNC GID*. Upon receipt of this message, *rcbp* invokes *buildWSList*() again. The function first get the maximum GID (*fromGid*) from the *WSList*, then retrieves all WSs whose GID is greater than *fromGid* but no greater than the received GID from message *MSG SYNC GID*. After than,

rcbp sends these WSs to *rcbr* in message *MSG RECOVERY DONE*. After this message is sent, the assistance of the recovery is done at the peer site. At the recovering site, upon receipt of the message, *rcbr* processes the WSs received along with the message like those in message *MSG TXN UPDATE*. After that, it sends message *MSG RECOVERY DONE* to *rmgrr*. Upon receipt of this message, *rmgrr* invokes $sendAllBufferedWS()$. This function first blocks the GCS channel, then sends all WSs in $WSBuffer$ to the replication backend of the recovering site. The replication backend then processes these WSs. After all WS have been processed, *rmgrr* unblocks the GCS channel, and the recovery is fully completed.

4.4 WriteSet Log

In theory, we do not need to store WSs separately. In PostgreSQL-R, update information of a transaction is stored in the Write Ahead Log (WAL). During a recovery, we can process the WAL and re-generate the WS structure. In this way, we don't need to keep redundant update information in the database. But there are drawbacks of this alternative. First, we need to scan the WAL to retrieve the update information and to construct the WS structure. This poses serious overhead to the recovery process. Second, PostgreSQL checkpoints the WAL from time to time to keep the log relatively small. We need to change the checkpoint criteria to force the WAL keep enough transaction records for the potential assistance of a recovery. Otherwise, the distributed recovery may be forced to use the TCS. Hence, we have looked at solutions not using the WAL.

Upon a transaction commit, a record containing the GID and the WS of the transaction needs to be logged. There appears to be three ways to implement the

logging mechanism. First, log the record directly in a file using Unix I/O interface. Second, log the record into a regular table of PostgreSQL-RR. Third, log the record into a system catalog of PostgreSQL-RR. Since we were not able to execute SQL queries from within PostgreSQL-RR itself, which would be needed to insert and retrieve WS from a regular table, we only discuss the first and the third option in the following subsections.

4.4.1 Log using File

Transactions commit sequentially in PostgreSQL-RR along with the sequential increase of the GIDs for these transactions. A WS record has to be written to the log file upon the commit of its corresponding transaction together with the corresponding GID. No deletion of records needs to be done until the file becomes too big. Record retrievals are only needed during the assistance of a recovery at the peer site: the peer retrieves WSs of transactions missed by the recovering site, which requires sequential read from the file only. A sequential file sounds like a perfect candidate for the job. But there are some technical problems that need to be solved.

The first is the format of the WS record in the file. Before the WS structure can be transferred via the network, it needs to be serialized. This means what we want to put into the file is actually the serialized WS, which is basically a bit stream of zeros and ones. How to differentiate the GID from the WS within a record, and how to differentiate a record from another record requires a predefined data format.

Second, the performance of the file access becomes worse when the size of the file becomes larger and larger. To reduce the size of the file, we might want to compress a large WS record before it is put into the file, which is a common technique used

by database management systems to record variable size objects. Then, we need to add an attribute to the record indicating whether the WS was compressed or not.

Third, we need to write the WS record into the file before the transaction commits. Then we have to coordinate the write with transaction commit to guarantee the atomicity property of the transaction. That is, if the transaction commits, then WS should be transferred from a peer to the recovering site. If the transaction does not commit, it should not be transferred.

Fourth, we need to develop our own library functions to access records according to the predefined data format, which is very time consuming and error prone.

4.4.2 Log using System Catalog

PostgreSQL uses system catalogs to record meta-data of the database, and provides a interface for efficient access to the catalog using caching and indexing. A system catalog is like any other table stored in the database, thus any operations on the system catalog can be easily wrapped in a transaction. This eases the implementation of the WSL dramatically. Only after a transaction commits does its WS become visible in the system catalog.

Serialized WS are stored in the system catalog in the form of $BYTEA$. $BYTEA$ is the data type in PostgreSQL for Binary Large Object, or $BLOB$, which is defined in the SQL standard. $BYTEA$ data is of variable size, and PostgreSQL automatically compresses it before it is stored into the catalog to save storage space. When a WS is retrieved from the catalog, we need to check if we need to decompress it before we un-serialize it back to WS structure.

We have created a system catalog *pg writeset* to store WS records for PostgreSQL-RR. This table has two columns. The first one is of $INTEGER$ type and stores GIDs; the second one is of $BYTEA$ type and stores serialized WSs. An index is built on the column GID using a B-tree. B-tree is an index structure that is very useful for range queries. These are the queries we have to pose for the $buildWSList()$ function to retrieve WSs. We use PostgreSQL's system table interface to retrieve ranges of records using the B-tree.

4.5 Snapshot in PostgreSQL-RR

To use TCS in a recovery, we first take a snapshot of the peer, transfer it to the recovering site, and then install the snapshot at the recovering site. This is relatively simple with the built-in features of PostgreSQL. PostgreSQL provides two features to front-end users: *pgdump* and *pgrestore*. These two features were introduced to allow the system administrator to backup the database during system migration.

Pgdump takes a consistent snapshot of the database. This can be viewed as an atomic action: only transactions that committed before the invocation of *pgdump* are reflected in the snapshot even if there are transactions running concurrently to when the snapshot is being taken. PostgreSQL actually suggests to avoid executing new transactions during the execution of *pgdump*, because those transactions will be missed by the snapshot. In PostgreSQL-RR, we must be able to handle concurrent transactions to support online recovery. Thanks to the PCS recovery implemented in PostgreSQL-RR, the writesets of transactions missed by the snapshot can be applied after the snapshot is installed at the recovery site.

Pgrestore basically converts the snapshot into SQL statements and feeds them to PostgreSQL. *Pgrestore* must be executed with PostgreSQL running in non-replication mode, since no WS needs to be built and be propagated to other running sites.

CHAPTER 5
Evaluation

In this chapter, we illustrate some experiments we did on PostgreSQL-RR. The focus of these experiments is to give an estimation of the cost of the distributed recovery using the two strategies, which in turn helps us to develop the heuristic used in the hybrid distributed algorithm to choose the optimal strategy to minimize the recovery time.

5.1 Synthetic Data

5.1.1 Setup

We ran our experiments in a local area network using two PCs running RedHat Linux. Each PC has 1 GB RAM and two processors (3.0 Ghz with 512 KB cache). The database contains 20 tables. Each table was created using a similar schema. For instance, table t a was created using the following SQL statement:

```
CREATE  table t_a  (
    t_a_id       int4        constraint t_a_pk primary key,
    t_a_param1   char(80),
    t_a_param2   int4,
    t_a_param3   float8,
    t_a_param4   date
);
```

5.1.2 TCS Recovery

Recall that the basic idea of the TCS approach is that the peer site reads its entire local database into a snapshot and transfers it to the recovering site. Then the recovering site installs the snapshot into its local database. Hence, the major factor that affects the recovery time using the TCS approach is the size of the database. In order to evaluate the dependency on database size, we first performed off-line recovery using databases of different sizes.

The setup of the experiments is quite straight forward. For each experiment, we first started only the peer site S_p with empty tables indicating no recovery is needed. Then, we started some client processes, which then connected to S_p and inserted records into its tables. By inserting different numbers of records into the tables, we created databases of different sizes. After the database was loaded, we stopped the clients and specified S_p as the potential peer site in the configuration file for site S_j. Then we started S_j up in replication mode. S_j then used S_p as the peer site to perform recovery. Since S_j initially had an empty WSL, the TCS strategy was automatically chosen.

Figure 5–1 shows recovery time when we increase the database size up to 1 gigabyte. T *Recovery* represents the total recovery time. T *pgdump* represents the time it takes the peer site to take a snapshot of its database. T *scp* represents the time it takes to transfer the snapshot from the peer site to the recovering site using Linux shell command *scp*. T *pgrestore* represents the time it takes the recovering site to install the snapshot into its database.

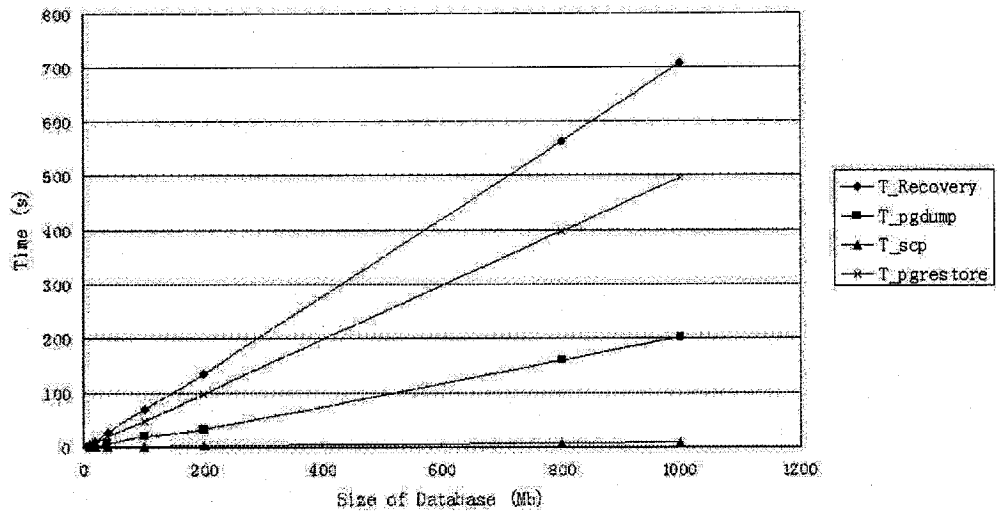

Figure 5–1: TCS Recovery

As we can see in the figure, it takes about 70 seconds to recover a 100 megabyte database, and it takes about 700 seconds to recover a one gigabyte database. Note that all time components in the figure increase linearly with the database size. Hence, we can extend the chart to estimate the recovery time for larger databases. For instance, it will take about 2 hours to recover a 10 gigabyte database, and it will take about one day to recover a 100 gigabyte database, which is quite expensive but still feasible. To recover a one terabyte database, it will take more than 8 days, which is not so practical.

Let us have a closer look at the components of the total recovery time. *T pgdump* counts for about 70 percent of *T Recovery* time. *Pgdump* is performed at the peer site, which might affect the peer during online recovery. Recalled that *pgdump* is a transaction that reads the entire database, and in PostgreSQL-RR, read operations

never block any other concurrent operations. *pgdump* does use some CPU time of the peer site, but it represents only one database client request, and hence the effect it has on the peer site should be reasonably small. *Pgrestore* is performed at the recovering site, hence has no effect on the existing system at all. T *scp* is so small that we could just safely ignore it.

5.1.3 PCS Recovery

The basic idea of the PCS approach is to transfer all WSs of transactions missed by the recovering site during its down time from the peer site and apply them into the database of the recovering site. Hence, the major factor that affects the recovery time using the PCS approach is the total number of WSs to be transferred and applied. Hence, we ran off-line recovery requiring different numbers of WSs to be transferred.

To run an experiment, we first started site S_p and S_j in replication mode indicating no recovery is needed. S_p and S_j had identical data where each database table had 10000 records lead to a database of 20 megabytes. After that, we started some client processes, which then connected to either S_p or S_j and submitted transactions. Each of these transactions contained one to three write operations, each write operation updating two records. After a while, we manually crashed S_j and terminated the clients who were connected to it. Clients who connected to S_p continued to submit transactions to S_p. When the desired number of transactions had been submitted to S_p after the failure of S_j (determining the WSs to be transferred), we terminated all clients. Then, we specified S_p as the potential peer in the configuration file and started S_j up again in replication mode. Since the WSL of S_j was not empty in

this case, the *Nthreshold* determined which recovery strategy to be used. We set *Nthreshold* to infinity to force a PCS recovery.

Figure 5–2 presents the time needed for recovery depending on the number of WSs to be transferred during recovery. $T\ Recovery$ represents the total recovery time. $T\ WSRetrieval$ represents the time it takes the peer site to retrieve all WSs missed by the recovering site from its WSL. $T\ WSApply$ represents the time it takes the recovering site to apply all updates that are captured in the WS into its database. $T\ Rest$ represents all other minor time components of the total recovery time, which includes the time for the GID synchronization, the time it takes to transfer WSs over the network, and the time it takes to process the buffered WSs. As we can see in the figure, $T\ WSApply$ counts for more than 90 percent of the total recovery time.

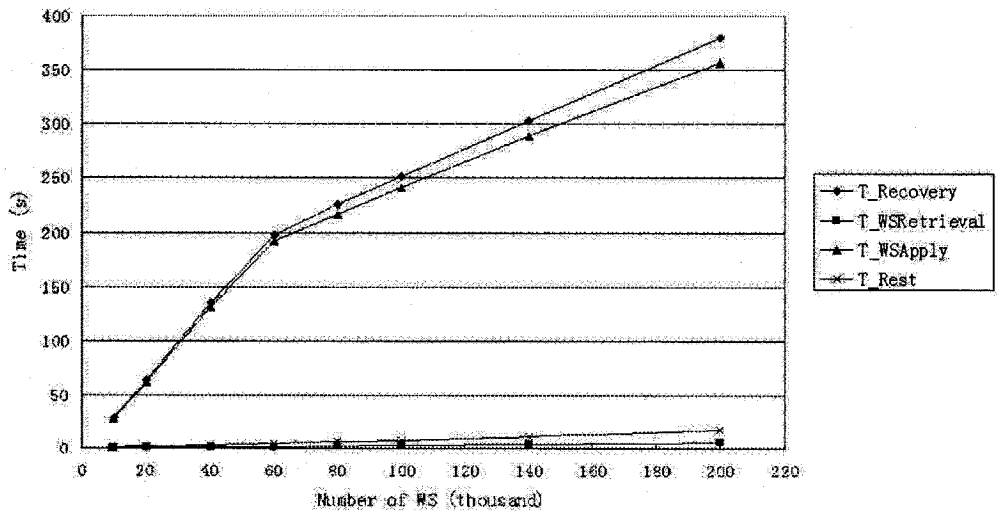

Figure 5–2: PCS Recovery

As shown in Figure 5–2, the relation between T *Recovery* and the number of WSs (N) is linear when N is less than 60000. If we consider the relation between T *Recovery* and N where N is greater than 60000, T *Recovery* increases proportioned to the increase of N again but at a lower rate. This can be explained with PostgreSQL's caching mechanism. When WSs are applied at the recovering site, some records may be accessed over and over again, and hence are cached by PostgreSQL. When the cache is loaded up, subsequent accesses to these records are more efficient. Of course, whether the recovery can take advantage of such mechanism depends on the database size.

We tested PCS recoveries that transferred and applied up to 200000 WSs where each of them contained 2 to 6 records. It takes about 4 minutes for a site to recover which missed 100000 transactions during its down time. And it takes about 6 minutes for a site that missed 200000 transactions to recover. As we can see, the PCS recovery is quite efficient. Again, we can extend the chart to estimate T *Recovery* for greater N. For example, it will take about 700 seconds for a site to recover where N is equal to 380000. In two hours, a site that missed about 4 million transactions can recover. In one day, a site that missed about 48 million transactions can recover. To get the numbers for a given application, such test runs should be made with the particular application in question.

One interesting number we want to have is the number of WSs that can be transferred and applied in one second, because this will determine if online recovery is possible. If during online recovery the remaining system commits more transactions per second than can be transferred with PCS to the recovering site, the recovering

site will never catch up. That is, recovery must be faster than the system commits new transactions. From the figure, when N is greater than 60000, $T\ Recovery$ = 0.0018N+20. So, in average, about 550 transactions can be transferred and applied per second.

Finally, let us have a look at the components. $T\ WSApply$ counts for more than 90 percent of the total recovery time, but being an action performed at the recovering site, $applyWS$ has no effect on the peer site during online recovery. Writeset retrieval is performed at the peer site, but since the $T\ WSRetrieval$ only counts for one percent of the $T\ Recovery$ time, we can safely ignore its effect on the peer if recovery is performed while transactions continue to execute on the peer.

5.1.4 Further Discussion

After we have the two charts presented in the previous two sections, we have a way to determine *Nthreshold*. We first measure the size of the database in question to get the corresponding $T\ Recovery$ time from Figure 5–1. Assume that we have $T\ Recovery$ time equals to t_1 in this case. We use t_1 in Figure 5–2 as the $T\ Recovery$ to find out the corresponding N, the number of missed WSs that can be transferred in the same time. This is the *Nthreshold*. It means that we can transfer and apply N WSs in the same time we can recover the site using the TCS approach. If the number of transactions missed by the recovering site is less than this N, then TCS recovery should be performed. Otherwise, PCS should be used. Now assume the application runs 100000 update transactions per hour and a site is down for two hours, i.e., 200000 transactions have to be transferred. In our configuration above,

for a database smaller than 500 megabyte, TCS will be used. Otherwise, PCS will be used.

5.1.5 Online Recovery

So far, we only discussed results from offline recovery experiments. In order to extend the above results to online recovery, we ran further experiment to explore how the ongoing execution of transactions affects the cost of recovery.

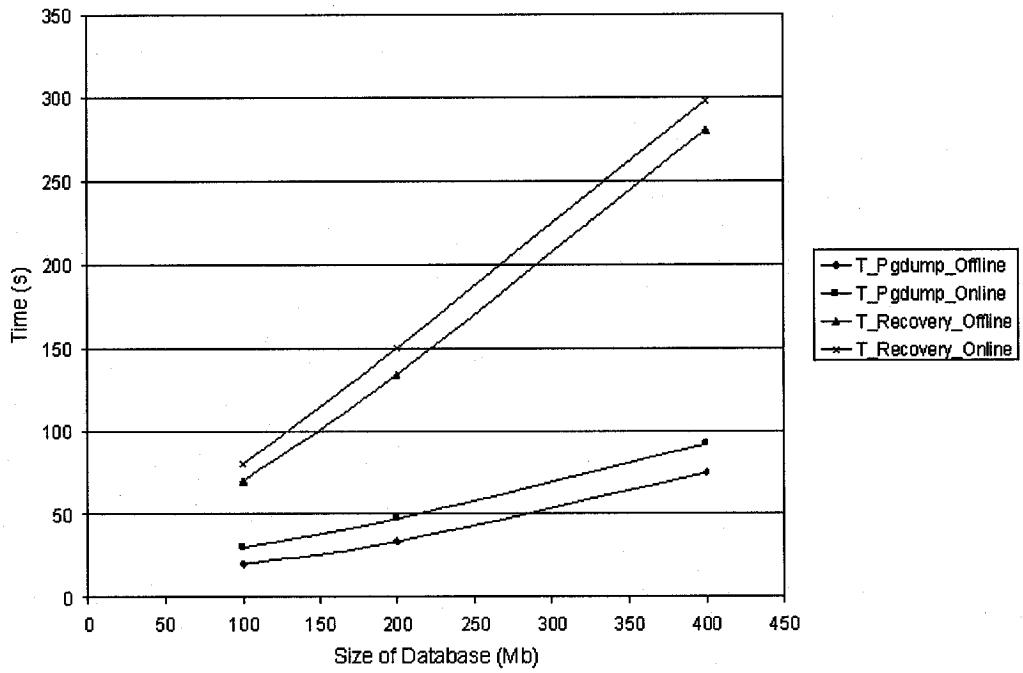

Figure 5–3: Online TCS

We first did experiments for online TCS recovery. For that, we first started two sites with six clients connected to each. Each client submitted 45 transactions per second (tps) to the server, hence, the total system throughput is 280 tps. Then

we started the third site S_j and forced it to recover using TCS. Figure 5–3 shows T *pgdump* and T *recovery* for both offline recovery and online recovery while increasing database size. As we can see, with online recovery, T *pgdump* needs more time than with offline recovery, since *pgdump* competes with other transactions on the peer site for CPU and disk resources. However, the effect on T *recovery* is minor. For instance, when the size is 400 Mb, T *Recovery Online* is 297 seconds, which is 17 seconds more than its counterpart T *Recovery Offline*, an 6 percent increase. However, the effect of recovery on the transactions executing on the peer site was quite significant. In our setup, the response times were 5ms before recovery and arose to 10ms during the execution of *pgdump*.

We also did experiments for online PCS recovery. We barely observed any increase in recovery time compared to offline PCS recovery. This supports our previous reasoning.

5.2 Benchmark Data

It is always good to see the performance of the system on a realistic application. We used the Open Source Development Lab's Database Test 1 (OSDL-DBT-1) kit [21], which is a deviation of the TPC-W benchmark [11] and simulates the on-line bookstore workload. There are 12 tables that record the customer information, the book information and the order information. The size of the database is determined by the number of customers and the number of items we choose for the test. The workload can be determined by choosing a different mix of browsing and ordering transaction. The system throughput is determined by the number of users simulated by the driver and the *thinktime* used by each user.

5.2.1 Setup

We ran our experiences in a local area network using two PCs running 2.6.12.2-smp Linux Kernel. Each PC has 512MB RAM and two Pentium III CPUs at 733.86 MHz. The database is built according to the requirements of the benchmark. We chose the browsing profile that about 80 percent of the transactions are browsing transactions and the rest are ordering transactions. We chose *thinktime* to be 3.0 seconds. The benchmark driver simulates 100 clients and maintains 20 connections with one database server for the clients. This database server is also served as the peer during recovery. The other database server is the recovering site.

5.2.2 TCS Recovery

The test is very similar to the one using synthetic data, except this time, the clients are controlled by the benchmark test kit. We tested the TCS recovery both off-line and online for three different database sizes. We recorded *T pgdump* and *T pgrestore*. *T pgdump* represents the time it takes the peer site to take a snapshot of its database. *T pgrestore* represents the time it takes the recovering site to install the snapshot into its database. We also recorded *T resp*, the response time clients experience when submitting *Home* transactions, both with and without the system undergoing a recovery.

Table 5–1: TCS Recovery

Size(Mb)	*T pgdump* (Off-line)	*T pgdump* (Online)	*T pgrestore*	*T resp*(ms) (w/o recovery)	*T resp*(ms) (with recovery)
100	9	12.3	39	106	131
210	18.7	35.1	87	319	458
355	41.3	76	196	334	611

As shown in Table 5–1, it takes 12.3 seconds to take a snapshot online for a 100 megabyte database, and 76 seconds for a 355 megabyte database. The overall performance of the system recovery is pretty good. The problem is that T *pgdump* increases more than linear with the increase in database size. The reason for this is the *pgdump* operation needs to read the table into the memory and then writes it to a file. As the size of the database goes up, the size of each table also goes up, and it will be harder and harder to fit the whole table into the available memory.

The transaction response time (T *resp*) increases with the size of the database with and without recovery. Without recovery, the response time with a 355 megabyte database is three times as high than with a 100 megabyte database. With recovery running in the background, the response time with a 355 megabyte database is four and a half times as high than with a 100 megabyte database. In particular, with a small database of 100 megabyte response times without and with recovery are nearly the same, while with a 355 megabyte database response times during recovery are nearly double as high as when no recovery process is running. When the database is small, nearly all data fits into main memory, and the chosen throughput of X does not put a high burden on the CPU. Hence, recovery and transaction processing do not compete, and recovery does barely affect response times. However, with a large database, a throughput of X leads to a higher load for CPU and memory, and recovery negatively influences transaction response time.

5.2.3 PCS Recovery

Similar to the tests using synthetic data, we recorded $T\ WSRetrieval$, $T\ WSApply$ and $T\ Recovery$ during both off-line and online PCS recovery. $T\ Recovery$ represents the total recovery time. $T\ WSRetrieval$ represents the time it takes the peer site to retrieve all WSs missed by the recovering site from its WSL. $T\ WSApply$ represents the time it takes the recovering site to apply all updates that are captured in the WS into its database. We also recorded the transaction response times at the clients.

Table 5–2: PCS Recovery

	Off-line		
N	$T\ WSRetrieval$	$T\ WSApply$	$T\ Recovery$
6105	0.1	22.5	23.5
15609	0.3	65.1	67.9
30538	0.6	146.0	151.6
	Online		
N	$T\ WSRetrieval$	$T\ WSApply$	$T\ Recovery$
6105	0.3	23	24.4
15609	1.7	67.1	78.1
30538	3.3	143.2	152.5

Table 5–2 shows the results for different numbers of WSs (N) that have to be transferred. $T\ Recovery$ increases linearly with the number of WSs to be transferred from the peer site to the recovering site. $T\ WSApply$ accounts more than 90 percent of the total recovery time. As described in previous sections, $applyWS$ is performed at the recovering site, hence has no effect on the peer site during recovery. WS retrieval is performed at the peer site, but $T\ WSRetrieval$ accounts for less than 2 percent of the total recovery time. So, we believe that PCS recovery has only minor

effect on the ongoing database system. Transaction response time experienced by the clients is the same during recovery and when no recovery takes place, which also implies PCS is very good for online recovery.

CHAPTER 6
Special Failures

To make our discussion complete, I want to address two more questions: how PostgreSQL-RR handles network partition and total failure.

6.1 Network Partition

As mentioned in a previous section, Spread implements EVS to provide uniform reliable messaging service even when the network is partitioned. This comes with both benefits and drawbacks. The uniform reliable messaging service enables PostgreSQL-RR to maintain data consistency even during site failures. But the failure model of EVS allows a communication group to partition and remerge at a later time. This allows any two nodes of the communication group in PostgreSQL-RR to commit two different sets of transactions and cause data to become inconsistent across the system. So, network partition is not handled correctly in PostgreSQL-RR.

We can extend PostgreSQL-RR to allow it to keep data consistent across the system even when network partition occurs. We borrow the idea from VS model. Only group members in the primary partition can proceed, any member in a non-primary partition must stop proceeding and emulate failure. At remerge the members that come from a non-primary partition have to perform recovery. There may at most be one primary partition in the system. The problem becomes how to determine the primary partition. There are two typical approaches: *majority* and *dynamic voting*. Determining the primary partition is a challenging task. There exist systems that

produce a primary partition layer on top of a GCS [12, 4]. Upon view change, this layer determines whether the new view is primary. It forwards the view message to the upper layer (e.g., to the communication manager of PostgreSQL-RR) indicating whether the view is primary or not.

6.2 Total Failure

The uniform reliable messaging service guarantees that any message delivered at a failed site is also delivered at all other running sites, which implies that data kept stored at each site is consistent to each other even in failure situation. That is, a failed site has always a prefix of transactions committed at running sites. In case of a total failure of the system, i.e., all sites crash (e.g., power-outage in a cluster), there is a way in which PostgreSQL-RR can recover from it. The system administrator first starts each site in non-replication mode and retrieves the largest GID (maxGid) from *pg writeset*. MaxGid is the GID of the last transaction that commits just before the failure of the site. After that, the system administrator sorts them in a list and restarts each site according to its order in the list. The first site starts up as a regular site without distributed recovery. The second site starts up as a recovering site with the first one as the peer. When the recovery is done, both sites are regular sites. Then, the administrator starts up the next in the list as a recovering site. When all sites in the list have recovered, the whole system is up. In theory, the system is ready to take requests from clients after the first site is up. The recovery of the recovering site does not prevent the peer or any other regular site from processing clients' requests.

CHAPTER 7
Conclusion

This thesis presents the design and the implementation of the hybrid distributed recovery algorithm that takes the best out of different recovery strategies. A site can be recovered by transferring either the whole database from a running site of the system to the recovering site (i.e., the TCS strategy), or only the update information of transactions missed by the recovering site during its downtime (i.e., the PCS strategy). The algorithm handles the case where a new site joins a running system as well as the case where a crashed site rejoins the system, and dynamically chooses the optimal one based on the feasibility and the estimation of the recovery cost. This thesis further presents how to recover PostgreSQL-RR after a total failure.

References

[1] Group Communication. Special Section. *Communications of the ACM*, 39(4):50–97, April 1996.

[2] M.E. Adiba and B.G. Lindsay. Database snapshots. In *VLDB*, pages 86–91. IEEE Computer Society, 1980.

[3] R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier applications. In *ICDE'02: Proceedings of the 18th International Conference on Data Engineering*, page 543, Washington, DC, USA, 2002. IEEE Computer Society.

[4] A. Bartoli and B.Kemme. Recovering from Total Failures in Replicated Databases. Technical Report 2003, available from http://webdeei.univ.trieste.it/Archivio/Docenti/Bartoli/repExt.pdf.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 1–10, New York, NY, USA, 1995. ACM Press.

[6] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP'87)*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[7] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with ISIS Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[8] M. Chouk. Master-slave replication, failover and distributed recovery in postgresql database. Master's thesis, McGill University, Montreal, Canada, 2003.

[9] Oracle Corporation. Oracle 9i Replication, June 2001.

[10] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems. Concepts and Design. 3rd edition.* Addison Wesley, Reading, MA, 2000.

[11] T.P.P. Council. Tcp benchmark w, 2000.

[12] D. Dolev, I. Keidar, and E.Y. Lotem. Dynamic voting for consistent primary components. In *PODC '97: Proceedings of the sixteenth annual ACM Symposium on Principles of Distributed Computing*, pages 63–71, New York, NY, USA, 1997. ACM Press.

[13] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64–70, 1996.

[14] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD Conf.*, 1996.

[15] R. Guerraoui and A. Schiper. Transaction model vs Virtual Synchrony model: bridging the gap. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, volume LNCS 938, pages 121–132. Springer, 1994.

[16] L. Irún-Briz, F. Castro-Company, A. Calero-Monteagudo F. García-Neiva, and F.D. Munoz-Escoí. Lazy Recovery in a Hybrid Database Replication Protocol. In *XII Jornadas de Concurrencia y Sistemas Distribuidos*, pages 295–307.

[17] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the The 18th International Conference on Distributed Computing Systems(ICDCS'98)*, page 156, Washington, DC, USA, 1998. IEEE Computer Society.

[18] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Int. Conf. on Very Large Databases (VLDB)*, 2000.

[19] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3), 2000.

[20] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2001.

[21] Open Source Development Lab. Descriptions and documentation of osdl-dbt-1, 2002.

[22] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *Proc. of the 14th IEEE Conf. on Distributed Computing Systems*, pages 56–65, June 1983.

[23] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[24] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks(DSN)*. IEEE Computer Society Press, 2001.

[25] PostgreSQL. homepage: http://www.postgresql.org/.

[26] R. Van Renesse, K.P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[27] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. IEEE Conf. Distributed Computing Systems(IDCDS)*, pages 561–568, 1993.

[28] Spread. homepage: http://www.spread.org/.

[29] UML. homepage: http://www.uml.org/.

[30] S. Wu. Integrating synchronous update everywhere replication into postgresql based on snapshot isolation. Master's thesis, McGill University, Montreal, Canada, 2004.