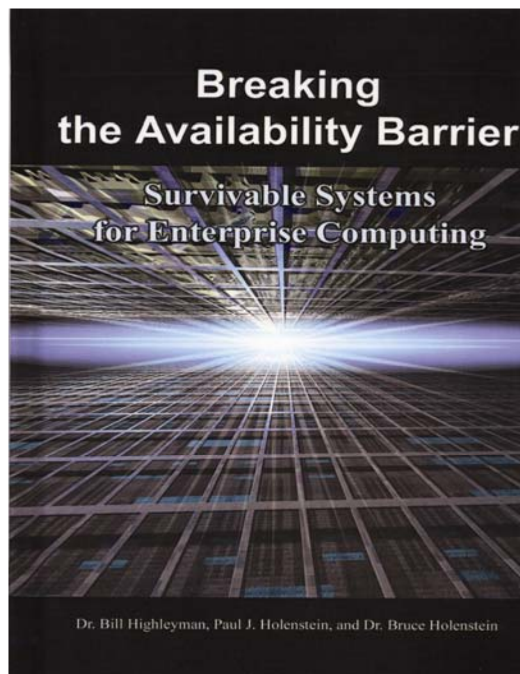


Breaking the Availability Barrier I
Survivable Systems for Enterprise Computing

Volume 1 of 3 Volume Series

Dr. Bruce Holenstein, Dr. Bill Highleyman, and Paul J. Holenstein

© 2007 Gravic, Inc. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the authors.

ISBN: 978-1-4107-9231-0 (e-book)
ISBN: 978-1-4107-9232-7 (Paperback)
ISBN: 978-1-4107-9233-4 (Dust Jacket)
Library of Congress Control Number: 2003099708

All products mentioned in this book are trademarks of their respective owners. The information in this book is provided on an as-is informational basis. The authors, owners, and publisher disclaim liability for any errors or omissions. The reader accepts all risks associated with the use of the contents of this book.

About the Authors:

The authors of the book, Dr. Bill Highleyman, Paul J. Holenstein, and Dr. Bruce Holenstein, have a combined experience of over 90 years in the implementation of fault-tolerant, highly available computing systems. This experience ranges from the early days of custom redundant systems to today's fault-tolerant offerings from HP (NonStop) and Stratus.

Click for Book Order Information: Breaking the Availability Barrier
or visit Amazon.com or Authorhouse.com to purchase.

Chapter 4 - Synchronous Replication

In Chapter 2, "*System Splitting*", we looked at the availability advantages of replicating or splitting a processing system. If a system is split into multiple independent nodes, it is important to have two or more copies of the database distributed across the network in order that application data is available to surviving nodes in the event of a node failure.

These database copies must all be kept in synchronism so that all applications running on all nodes will have the same view of the application data. This synchronism can be easily achieved via asynchronous data replication – there are many data replication products available today that provide this capability without having to modify the applications. However, if all nodes are actively participating in the processing of transactions, data collisions may occur at an uncomfortable rate due to replication latency. These data collisions, unless detected and resolved, will result in database contamination.

Data collisions can be avoided by synchronously updating all database copies so that all copies of all data items are always guaranteed to be the same. This can be accomplished with synchronous data replication, which is the subject of this chapter.

Replicating Systems

The replication of systems is an important and popular technique to ensure that critical computing systems will survive system failures caused by anything from component failures to man-made or natural disasters.

Under a typical system replication scenario, one system is the primary and handles the entire transaction load. As shown in Figure 4-1a, updates made by the primary system to its database are replicated to the backup system. The backup system is passive except for perhaps supporting read-only operations such as query and reporting.

As shown in Chapter 1, "*The 9s Game,*" fully replicating a system, in addition to providing protection from disasters, has the effect of doubling its 9s, thus dramatically improving the system's availability.

Splitting Systems

Alternatively, significant availability advantages can be achieved by simply splitting a single system into *k* nodes. In Chapter 2, we showed that doing so could increase system reliability (i.e., increase its mean time before failure, MTBF) by more than a factor of *k*.

However, when we split a system (for example, splitting a 16-processor system into four 4-processor nodes), all nodes must be actively sharing the load. This implies that all nodes are updating the database.

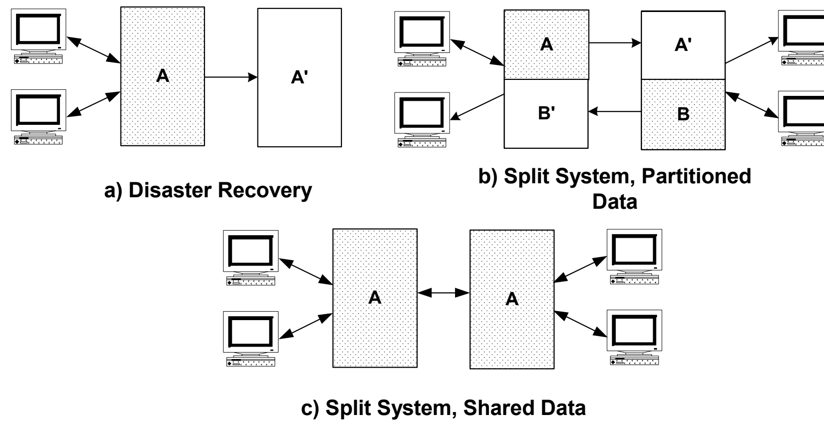
Often, the database can be partitioned so that only one system can update a given partition; and those updates can be replicated to the other systems for read-access only (Figure 4-1b). In this case, the most serious concern is *replication latency*, or the time that it takes for an update to propagate from the source node to the target node. Updates in the replication pipeline may be lost in the event of a system failure.

However, in the more general case, any system in the network must be able to update any data item (Figure 4-1c). Those updates then must be replicated to the other databases in the network. We call these *active/active* replication applications. In addition to the problems imposed by replication latency, as described above, active/active applications present additional significant problems. One

of the most severe problems is data collisions. To the extent that there is replication latency, there is a chance that two systems may update the same data item in different copies of the database simultaneously. These conflicting updates then will be replicated across the network and result in a corrupted database (i.e., the value of the data item will be different in different instances of the database).

For instance, an application at node A might change the value of a particular data item from 10 to 15. At nearly the same time, an application at node B might change that same data item from 10 to 20. Node A will then replicate its value of 15 to Node B, which will set the data item value to 15. Likewise, Node B will replicate its value of 20 to node A, which will set the value of the data item at its node to 20. Now the data item not only has different values at the two nodes, but both values are wrong.

We call these simultaneous conflicting updates *data collisions.* Data collisions must be detected and resolved, often manually.



a) Disaster Recovery

b) Split System, Partitioned Data

c) Split System, Shared Data

Split System Architectures
Figure 4-1

Data Collisions

The probability that data collisions will occur is surprisingly high. An extension of Jim Gray's work[22], presented later in Chapter 9, "*Data Conflict Rates*," shows that the data collision rate in such a network is given by

$$\text{Data Collision Rate} = \left(\frac{d-1}{d}\right)\frac{(kra)^2}{D}(L + t/a)$$

where

- *r* is the transaction rate generated by one node in the network,
- *a* is the number of replicable actions in a transaction (updates, inserts, deletes, and in some systems read/locks),
- *t* is the average transaction time,
- *k* is the number of nodes in the network,
- *L* is the replication latency time,
- *d* is the number of database copies in the network,
- *D* is the size of the database (in terms of data objects – i.e., the lockable entities).

Consider a system split into four nodes with a requirement to maintain an up-to-date database at each node (*k*=d=4). Let us assume that each node generates a leisurely ten transactions per second (*r*=10) and that an average transaction involves four updates (*a*=4) and requires 200 milliseconds to complete (*t*=.2). Furthermore, let us assume that our database requires 10 gigabytes and that an average row (the lockable entity) takes 1,000 bytes. Thus, the database contains 10 million lockable objects (D=10,000,000). Finally, the replication latency is 300 milliseconds (L=.3).

[22] Gray, J.; et al.; "*The Dangers of Replication and a Solution*," ACM SIGMOD Record (Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data), Volume 25, Issue 2; June, 1996.

Using the above relation, we find that our system will create over 2.4 collisions per hour. This can be a major headache. If the nodal transaction rate increases to 100 transactions per second, the collision rate jumps to over 240 collisions per hour. This will certainly keep a team of people busy. If the system then grows to eight nodes, the collision rate will explode to over 1,100 collisions per hour. This is untenable.

Data collisions first must be detected and then must be corrected either manually or by automatic conflict resolution via business rules. Collision detection methods not only add overhead to the replication engine, but they also are only the start to collision resolution. The correction and resynchronization of the database is often a lengthy and manual operation since automated resolution rules are often not practical.

Synchronous Replication

In this chapter, we deal with methods for avoiding data collisions in active/active applications rather than having to correct them.[23] By implementing collision avoidance, there is no need for a collision detection mechanism; nor is there need for a resolution strategy that may involve complex business rules.[24]

The avoidance of collisions requires that all data items be updated synchronously. That is, when one copy of a data item is updated, no other copies of that data item can be changed by another update until all other copies have been similarly updated as well. We call this *synchronous replication.*

Synchronous replication carries with it a performance penalty since the transaction in the originating node may be held up until all

---

[23] Holenstein, B. D.; et al.; "*Collision Avoidance in Data Replication Systems*," United States Patent Application No. 2002/0133507; Sept. 19, 2002.

[24] This is an oversimplification. As pointed out later, network failures and node failures require resynchronization of the databases in order to recover. However, this statement is valid for normal operations.

data items across the network have been updated. In this chapter, we explore the performance impact of synchronous replication.

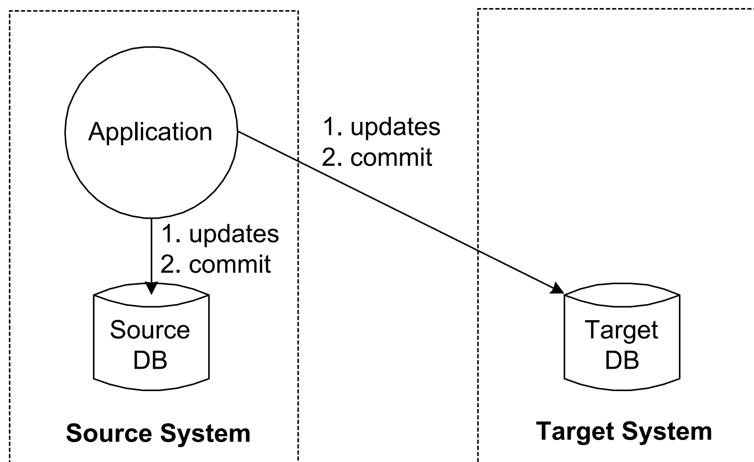
We consider two techniques for synchronous replication – dual writes and coordinated commits. To simplify the analysis, we consider a system comprising only two nodes. This analysis is extended later to systems comprising multiple nodes, and the conclusions are the same.

Dual Writes

The term *dual writes* is just another name for network transactions. Dual writes involve the application of updates within a single transaction to all replicates of the database. This is accomplished by using a two-phase commit protocol under the control of a distributed transaction manager. The transaction manager ensures that all data items at all sites are locked and are owned by the transaction before any updates are made to those data items, and it then ensures that either all updates are made (the transaction is committed) or that none of them are made (the transaction is aborted). In this way, it is guaranteed that the same data items in different databases will always have the same value and that the databases therefore will always be consistent.

A simple view of synchronous replication using dual writes under a transaction manager is shown in Figure 4-2. A transaction is started by the application, and updates are made to the source database and also to the target database across the network (1). The updates to the target database may be generated directly by the application, may be generated by an intercept library, or may be generated by database triggers that invoke, for instance, a stored procedure when an update is made. Each updated data item is locked, and the locks are held until the completion of the transaction. When all updates have been completed, the transaction will be committed. At this time, the transaction manager will apply all updates to the source and target databases (2). If the transaction manager is unsuccessful in doing this, then the transaction is aborted; and no updates are made.

There is a very small window of confusion that may cause the outcome of a transaction to be uncertain under some failure conditions. This is a characteristic of the two-phase commit protocol commonly used by transaction managers. Typically, when the originating application is ready to commit, the transaction manager will ask each node involved in the transaction if it is ready to commit. If all nodes concur that they are holding all locks and have safe-stored all modifications (completion of phase 1), then the transaction monitor will command all nodes to commit (phase 2). Should the network fail after a remote node has responded favorably to the phase 1 "ready to commit?" query but before that remote node has received the phase 2 commit command, then the remote node does not know whether the transaction was committed or aborted. This situation must be resolved either manually or by business rules governing the system behavior in this very unlikely situation.



Dual Writes
Figure 4-2

Note that transaction updates under a transaction manager may be done either serially or in parallel. If the updates are serial, then the application must wait not only for the local updates to be made, but it must also wait for the update of each remote data item over the network. If updates are done in parallel, then to a first approximation

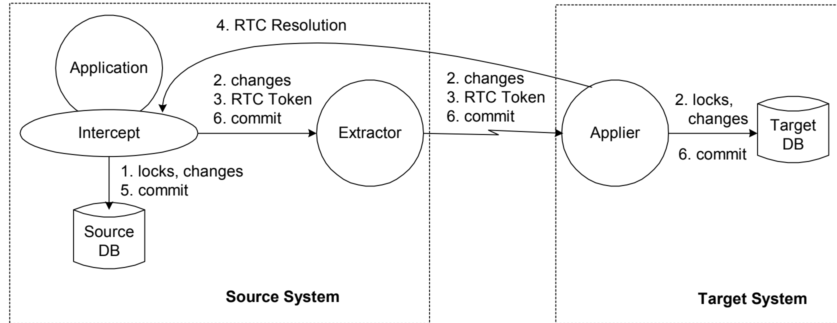
the application is delayed only by the communication channel propagation time. The read/write time at the target database is transparent to the application since it must spend this same amount of time updating the source database.

Most operating systems or databases today do not support dual writes directly. Therefore, to implement dual writes, it is usually necessary to actually modify the application, to bind in an intercept library, or to add triggers to the database to perform the writes for each data item to each database copy. Thus, implementing a dual write solution is invasive to the application in most cases.

Coordinated Commits

An alternative approach to dual writes is to begin independent transactions on each node and then to *coordinate the commits* of those transactions so that either they both commit or that neither commits. In this case, normal data replication techniques (see Chapter 3, "*Asynchronous Replication*") are used to propagate updates asynchronously to the target node. There they are applied directly to the target node's database as part of its transaction that was started on behalf of the source node. In this way, the propagation of updates over the network is transparent to the application.

A simplified view of an implementation for coordinated commits is shown in Figure 4-3. The application first will start a local transaction. As the application locks data items and makes updates to its source database (1), the changes to the database are captured (either by reading an audit file or log file or by intercepting the update commands from the application or the database). These updates are sent to the target node (2), where a transaction is started, locks are acquired, and the updates are made to the target database.



Coordinated Commits
Figure 4-3

When the application attempts to commit the transaction, the commit is intercepted (perhaps by an intercept library linked into the application). Before the source node is allowed to commit the transaction, a Ready-To-Commit (RTC) Token is sent to the target node (3) through the replicator to assure that the token will arrive at the target node after the last update. The RTC Token queries whether all of the changes to the target database are ready to be applied. At this point, the target node will respond to the source node with an RTC Resolution message (4). The source node side of the data replicator will release the commit to the source database (5). When this has committed successfully, a commit message is sent to the target node (6), which then will commit the transaction.

Should the target node respond negatively to the RTC Token, then the transaction is aborted at the source node. There is the possibility that the source commit might succeed and that the subsequent target commit fails. Since the target node has guaranteed that it is holding locks on all of the data items to be updated when it returns the RTC Resolution, then this type of failure should occur only if the target node or the network fails after the target node has returned the RTC Resolution and before it receives the commit directive from the source node. This failure window is similar to the window of confusion described earlier for dual writes.

In this case, the normal data resynchronization capabilities of the replication engine will have to be invoked to resynchronize the databases. However, the databases will be out of sync anyway when the target node is returned to service. The lost transaction is just one of many that will be recovered through the resynchronization procedure. Resynchronization facilities are normally provided with data replication engines. However, this is often not the case if dual writes are being used – database resynchronization following a network or target node failure often requires a user-written resynchronization utility.

Note that coordinated commits are implemented with a standard asynchronous data replication engine that has been enhanced to coordinate the commits at the various nodes. Therefore, as with asynchronous replication, an application can be retrofitted to be distributed using coordinated commits without modifying the application. The use of coordinated commits, unlike dual writes, is non-invasive to the application.

In addition, data replication offers the option to enhance the efficiency of data communication channel utilization by buffering the many small messages involved and then sending them to the remote nodes as blocks of changes. When dual writes are used, each database change must be sent individually.

Application Latency

From a performance viewpoint, we are interested in the additional delay imposed upon a transaction due to having to wait for the completion of updates to the target node. We call this additional delay the *application latency* caused by synchronous replication.

Note that if data replication is asynchronous rather than synchronous, then the application will not be delayed by data replication. Instead, the target database will lag behind the source database by a time interval which we have previously called *replication latency.*

Also note that application latency will increase the response time for a transaction but will not in itself affect thruput. Thruput can be maintained simply by configuring more application processes to handle the transaction load. This solution, of course, assumes the use of well-behaved application models that allow the use of replicated application processes such as NonStop server classes.

Let us now calculate the application latency for dual writes and for coordinated commits so that they can be compared.

Dual Writes

To simplify our analysis of dual write application latency, we assume that all remote database operations are done in parallel with the database operations at the source node. We further assume that all database operations are full updates that require a read/lock of the data item followed by a write rather than simple operations such as read/locks (that is, fetches), inserts, and replacements that require only one access of the database. It is quite straightforward to modify the following relationships to account for these factors. In fact, we do so later in this chapter and will show that the general conclusions reached are unchanged.

In order to update a data item across the network, a read/lock command first must be issued and the data item then received. Next, the updated data item must be returned to the target database and a completion status received. Thus, there are four network transmissions required to complete one update.

In addition, the transaction manager's two-phase commit protocol requires four network transmissions. A prepare-to-commit message is sent and is followed by a response (phase 1). Then the commit message is sent and is followed by its response (phase 2). Only upon receipt of the commit response is the transaction considered to be complete at the source node.

Let

L_{dw} be the dual write application latency,
n_u be the average number of updates in a

transaction,

t_c be the communication channel propagation time, including communication driver and transmission times.

Then

$$L_{dw} = 4n_u t_c + 4t_c \qquad (4\text{-}1)$$

In Equation (4-1), the first term is the communication time required to send the updates to the target node. The second term is the communication time required to commit the transaction.

The above has ignored the processing time required for generating the additional remote database operations and for processing their responses. These times generally are measured in microseconds, whereas channel propagation times typically are measured in milliseconds.[25]

Coordinated Commits

Coordinated commit replication requires the use of a data replication facility to propagate the updates to the target node. Most of the time spent by this facility is invisible to the application providing that updates to the remote database are made as soon as they are received without waiting for the commit (the optimistic strategy).

However, once all updates have been made, the application then must wait for the RTC Token to be exchanged before it can carry out its transaction commit. Note that once the commit has completed at the source node, the subsequent commit at the target node is asynchronous relative to the application. The commit's success is guaranteed since the target node has acquired locks on the data items to be changed, and it will apply the changes upon receipt of a commit directive from the source node. Therefore, the source node does not

[25] For a more detailed analysis of processing times, contact the authors.

have to wait for the target node's commit to complete, and the target node commit thus does not add to application latency.

We estimate the application latency for coordinated commits, L_{cc}, as follows. Let

- L_{cc} be the application latency for coordinated commits,
- t_p be the processing delay through the replicator exclusive of the communication channel propagation time,
- t_c be the communication channel propagation time, as defined previously.

The RTC Token is sent through the replicator following the last transaction update to ensure that the token is received by the target node after the final update has been received. The time to propagate the RTC Token is therefore the replication latency of the replicator, t_p, plus the communication channel delay, t_c. The return of the RTC Resolution requires another communication channel delay. Thus,

$$L_{cc} = t_p + 2t_c \qquad (4\text{-}2)$$

Synchronous Replication Efficiency

Let us define a comparative measure of synchronous replication efficiency as the ratio of dual write application latency to coordinated commit application latency:

$$e = \frac{L_{dw}}{L_{cc}}$$

where

- e is comparative synchronous replication efficiency,
- L_{dw} is dual write application latency,
- L_{cc} is coordinated commit application latency.

Thus, for $e > 1$, coordinated commits outperform dual writes. For $e < 1$, dual writes perform better.

From Equations (4-1) and (4-2),

$$e = \frac{4n_u t_c + 4t_c}{t_p + 2t_c} = \frac{2(n_u + 1)t_c}{\frac{t_p}{2} + t_c} \qquad (4\text{-}3)$$

Equation (4-3) can be rewritten as

$$e = \frac{2(n_u + 1)}{1 + t_p / 2t_c} = \frac{2(n_u + 1)}{1 + 1/p} \qquad (4\text{-}4)$$

where

p is the round trip communication channel time expressed as a proportion of replication delay time:

$$p = 2t_c / t_p \qquad (4\text{-}5)$$

Figure 4-4 shows replication efficiency e plotted as a function of communication channel time p for various values of transaction sizes n_u. The regions of excellence for dual writes and for coordinated commits are shown.

The values of p and n_u for e=1 define the excellence boundary between dual writes and coordinated commits. From Equation (4-4), this relation is

$$2(n_u + 1) = 1 + 1/p$$

or

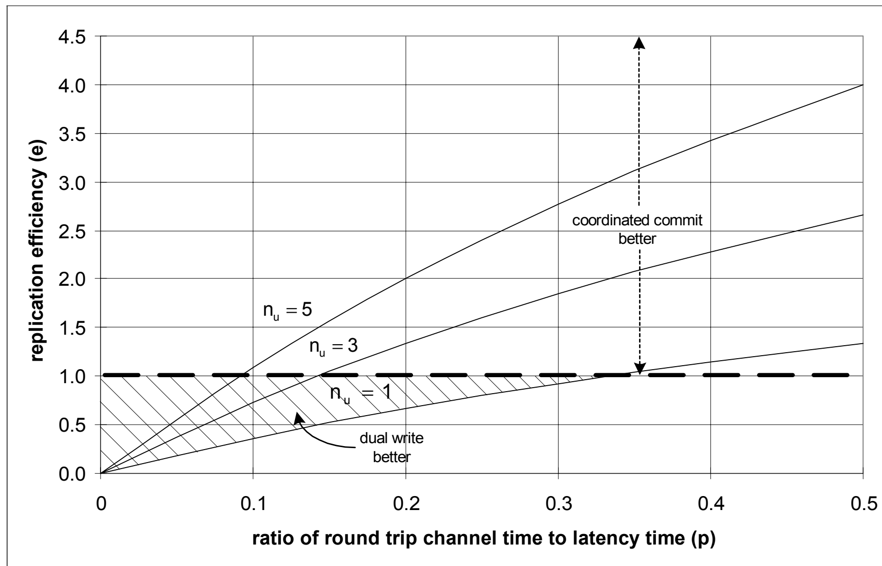
$$p = 1/(2n_u + 1) \qquad \text{for } e = 1 \qquad (4\text{-}6)$$

This relationship is shown in Figure 4-5.

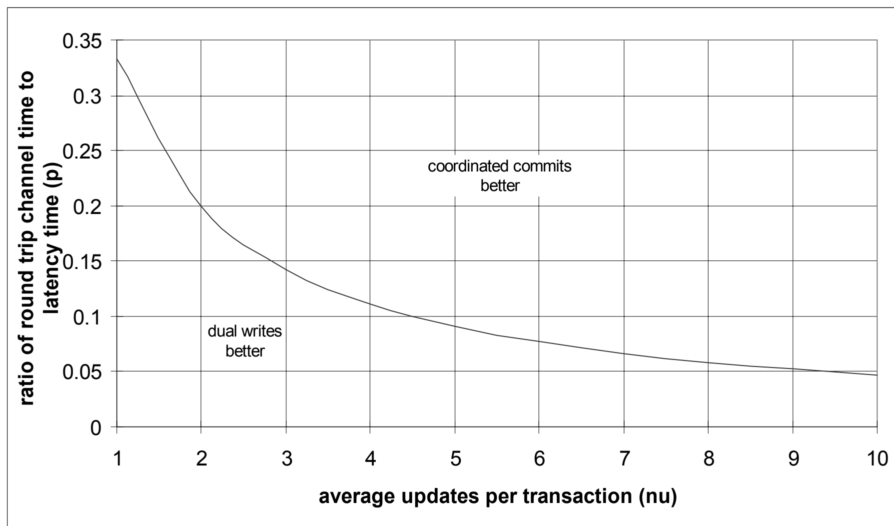
Figures 4-4 and 4-5 are for the case of parallel read/writes. Remember that the parameter, *p*, shown on the horizontal axis of Figure 4-4 and on the vertical axis of Figure 4-5, is the ratio of the round trip channel propagation time to the replication latency time. As such, it is a measure of the channel propagation time in terms of latency time. As a ratio, *p* is dimensionless. For instance, if the round trip channel propagation time is 20 msec., and if the latency time is 50 msec., then *p* is 0.4. If we reduce the round trip channel propagation time to 10 msec., then *p* is reduced to 0.2. That is to say, round trip channel time is 20% of the replication latency time.

The efficiency expressions for serial read/writes do not lend themselves to such simple charting. However, the efficiency for serial read/writes will be even better for coordinated commits since dual writes will have the additional application latency of having to wait for the remote reads and writes to complete.

Likewise, these figures are for the case of no simple database operations (operations that may require only a single communication channel round trip such as read/locks, inserts, replacements). To the extent that some updates are simple operations such as these, dual write performance will be better than shown since network traffic is reduced. Our efficiency curves may be modified easily to reflect this situation.



Synchronous Replication Efficiency
Figure 4-4



Equal Efficiency (e=1)
Figure 4-5

Scalability and Other Issues

There are other issues to consider when comparing synchronous replication algorithms, including scalability (as it relates to performance) as well as various other algorithm optimizations. We now discuss these issues, including the effects of multiple database copies, communication channel efficiency, and transaction profile.

Scalability

Multiple Database Copies

If there are d database copies in the application network, then an application using dual writes must make d modifications for every database modification required by the transaction. However, under coordinated commits, the application must make only one database modification for each required by the transaction; the other $(d - 1)$ modifications are made by the replication engine and do not affect the application.

Therefore, if other processing is ignored, application latency (which adds to transaction response time) under dual writes will increase with the number of database copies, d, whereas the transaction response time for coordinated commits is relatively unaffected by the number of database copies. Thus, coordinated commits are more scalable than are dual writes.

Communication Channel Efficiency

With dual (or plural) writes, each modification must be sent to all database copies as the modifications are made at the source node. Therefore, this method will send many small messages over the network. The coordinated commit method, on the other hand, has the opportunity to batch multiple change events into a single

communication block and thus send several change events within a single block.

For instance, if the application network is making 10 database modifications per second, then 20 messages per second must be sent over the network to each database copy by dual writes (assuming that all modifications require a read followed by a write). Coordinated commits may need to send only one or a few blocks, depending upon the replication latency desired.

If the system is making 1,000 modifications per second, then 2,000 messages per second must be sent to each database copy by dual writes. However, the number of messages sent by coordinated commits, even to achieve small replication latencies, can be one or two orders of magnitude less than this.

As system activity grows, network traffic due to dual writes will grow proportionately. However, network traffic generated by coordinated commits will grow much more slowly as more and more messages are accumulated into a single communication block before they must be sent. Therefore, coordinated commits are much more scalable with respect to communication network loading than are dual writes.

Transaction Profile

Another difference between dual writes and coordinated commits has to do with the transaction profile. In dual writes, the individual database accesses and updates required by a transaction are each delayed as they are sent over the communication channel to the target systems. At commit time, the application must wait for a distributed two-phase commit to complete.

In contrast, with coordinated commits, the source application transaction runs at full speed until commit time, and then pauses while the RTC is exchanged. At the end of this exchange, the application waits for a local (single node) commit to complete. This single node commit should be significantly faster than the distributed two phase commit required to ensure durability on the target systems.

Depending upon the application design, this can be quite advantageous, since the entire application latency is lumped into the commit call time. Furthermore, there may be no slowdown in those applications that support non-blocking (nowaited) commit calls since all of the coordinated commit application time will occur while the application is processing other work.

Read Locks

Coordinated commits and dual writes process read lock operations differently. At times, an application will read a record or row with lock, but not update it (for example, if an intelligent locking protocol is in use). With many implementations of dual writes, the target system record or row will be locked when the source record or row is locked. With coordinated commits, only the source data items are affected – read locks are not necessarily propagated. They only need to be propagated if the data item is subsequently updated. Therefore, not only will dual writes impose more overhead on the network and on the target system, but other transactions may be held up if they are trying to access the target data items that are locked. Thus, coordinated commits should support a higher level of concurrency than dual writes.

Other Algorithmic Optimizations

Of course, some optimizations to the dual write algorithm could be implemented. For example the individual I/O operations could be batched or aggregated and sent together. These optimizations effectively morph the dual write algorithm into a variant of the coordinated commit algorithm with some associated performance and network efficiency gains.

Examples

Geographically Distributed Systems

As an example, consider two systems, one in New York and one in Los Angeles, with the following parameters:

t_p	50 msec.	replication engine processing time, or the time to propagate an update through the replication engine from the source database to the target database, excluding communication channel propagation time.
t_c	25 msec.	communication channel propagation time, or the amount of time required for a message to propagate from the source node to the target node or vice versa over the provided communication channel, including line driver and transmission time.

These parameters result in a value for *p* (from Equation (4-5)) of 1.0. The resulting efficiency for a given transaction size is, from Equation (4-4),

$$e = \frac{2(n_u + 1)}{1 + 1/1.0} = (n_u + 1)$$

Efficiency as a function of the number of updates for this example is then:

n_u	e
1	2.0
2	3.0
3	4.0
4	5.0

As is seen, coordinated commits are more efficient for all transaction sizes in the above example. This is a direct result of the communication channel delays. The application latency due to synchronous replication via coordinated commits is, from Equation (4-2), 100 msec. The application latency under dual writes is this value multiplied by the efficiency factor, e, in the table above.

Generally, coordinated commits may be more efficient except when the communication times are very small. Note, however, that even at the speed of light, it takes a signal about 25 milliseconds to travel round trip between New York and Los Angeles. Data signals over land lines will take at least twice as long, or about 50 msec. for a New York – Los Angeles round trip. A London – Sydney round trip may take about 250 msec.

Consequently, a generalized statement is that dual writes are appropriate for campus environments with small transaction sizes. On the other hand, coordinated commits are appropriate for wide-area network environments or for large transactions. In addition, an application can be retrofitted to support coordinated commits without any recoding by installing an appropriate data replication facility. This facility usually brings with it auto-recovery of files that have become unsynchronized due to network or node failures, a capability which will probably have to be specially developed for dual writes.

In general, as can be seen from Figures (4-4) and (4-5) and from Equation (4-4), the larger the average transaction or the longer the communication channel propagation time, the more efficient coordinated commits become.

Adding to our rules from Chapters 1 through 3, we have

> **Rule 17:** *For synchronous replication, coordinated commits using data replication become more efficient relative to dual writes as transactions become larger or as communication channel propagation time increases.*

Collocated Systems

If the systems are collocated and interconnected via very high speed channels, the communication channel delay approaches zero; and dual writes may perform better for normal transactions.

However, for long transactions, there will come a point at which coordinated commits will be more efficient. This will be the point at which the processing time for the dual reads and writes at the target node, coupled with whatever communication channel delay exists, exceeds the time to exchange the RTC Resolution. Examples of long transactions are batch streams, box-car'd transactions, and database reorganizations.

To get a feel for this, consider the example of the previous section, but where the communication channel delay, t_c, is 2 msec. instead of 25 msec. In this case, p is .08; and efficiency from Equation (4-4) is

$$e = \frac{2(n_u + 1)}{1 + 12.5} = \frac{(n_u + 1)}{6.75}$$

Solving this for $e = 1$, we find that replication for transactions of six updates or more will be faster for coordinated commits than it will be for dual writes. For a six-update transaction, the coordinated commit application latency is, from Equation (4-2), 54 msec.

Efficiency Model Extensions

Dual Write Single Round Trip Operations

As pointed out earlier, not all database operations require two communication channel round trips as has been assumed so far. For instance, read/locks, replaces, and inserts require only one round trip – send the operation and receive the completion status. We can extend the above analysis very simply to account for these operations. Instead of there being $4n_u$ communication channel round trips for a transaction, there are $4n_{u2} + 2n_{u1}$ round trips, where

n_{u1} is the number of transaction operations that require one round trip,

n_{u2} is the number of transaction operations that require two round trips,

and $n_u = n_{u1} + n_{u2}$.

Let us define

$$n_u' = n_{u2} + n_{u1}/2 \qquad (4\text{-}7)$$

The previous equations now hold except that n_u is replaced with n_u'. The charts of Figures 4-4 and 4-5 still hold except for the substitution of n_u' for n_u.

Dual Write Serial Updates

If database changes are made serially when using dual writes, then the application must wait for all of the changes to be made not only to its local database but also to the remote database. Let us define t_o as the average database operation time for the particular mix of operations generated by a transaction. Then, in addition to the communication delays, the application will have to wait an additional time of $n_u t_o$ for the remote database operations to complete. The dual write application latency time, originally given by Equation (4-1), now becomes

$$L_{dw} = (4n_u' + 4)t_c + n_u t_o \qquad (4\text{-}1a)$$

and the efficiency relationship given by Equation (4-3) becomes

$$e = \frac{(4n_u' + 4)t_c + n_u t_o}{t_p + 2t_c} \quad (4\text{-}3a)$$

This can be written as

$$e = \frac{2(n_u' + 1) + \frac{n_u}{2}\frac{t_o}{t_c}}{1 + 1/p} \quad (4\text{-}4a)$$

where *p* was defined earlier as,

$$p = 2t_c / t_p$$

If remote database operation time, t_o, is very much less than the communication channel time, t_c, then Equation (4-4a) reduces to Equation (4-4). However, as the communication channel delay t_c approaches zero, Equation (4-4a) approaches

$$e \approx \frac{n_u t_o / t_c}{t_p / t_c} = n_u \frac{t_o}{t_p} \quad (4\text{-}8)$$

So far as estimating a reasonable value for the average database operation time, t_o, we note the following. Today's high-speed disks (15,000 rpm) have an average random access time of about 7 milliseconds (2 msec. rotational latency, 5 msec. seek time). If the record or row is in disk cache, today's systems require about 0.1 msec. to find it. If buffered writes are used (the write data is checkpointed to another processor and is physically written to disk some time later in the background), then a disk write will take about the same time as a disk read from cache (0.1 msec.), assuming that read and write cache hits are about the same (actually, disk writes might take a little longer due to block splits and index maintenance, but this is ignored here). A well-tuned OLTP application should be achieving something in the order of 80% cache hits. If this is the case, then 80% of read/write accesses will require about 0.1 msec. for cache

access; and 20% will require about 7 msec. for a physical disk access. This results in an average disk access time of about 1.5 msec.

If we assume typical values for t_o of 1.5 msec. and for t_p of 50 msec., we can see from Equation (4-4a) that serial operations make little difference in efficiency for large communication channel delays. As communication channel delays disappear, the efficiency factor *e*, rather than going to zero, reduces instead to the value given by Equation (4-8). But for our typical values (and assuming four operations per transaction, or n_u = 4), this asymptotic value is only .12. Thus, we can conclude that, in normal cases, dual write serial updates do not significantly affect the performance comparison between dual writes and coordinated commits.

Plural Writes

The above analysis for serial dual write updates assumed that there were only two copies of the database in the application network. But if there are more copies, then the application must wait for each operation to be completed on each remote database in turn. This means that the plural serial write application latency time becomes

$$L_{dw} = (4n_u' + 4)t_c + (d-1)n_u t_o \qquad (4\text{-}1b)$$

where

d is the number of databases in the application network.

The relative efficiency is then

$$e = \frac{2(n_u' + 1) + (d-1)\frac{n_u}{2}\frac{t_o}{t_c}}{1 + 1/p} \qquad (4\text{-}4b)$$

which approaches

$$e \approx (d-1)n_u \frac{t_o}{t_p} \qquad (4\text{-}9)$$

as communication channel time t_c approaches zero.

If updates are done in parallel, then the original analysis still holds.

Deadlocks

A potential problem with synchronous replication whether it is done via dual writes or coordinated commits, is the possibility of a deadlock. A deadlock occurs when two different applications must wait on locks held by the other. In non-distributed applications, this can occur if the applications are trying to lock two different data items but in different order. Application 1 locks data item A and tries to get the lock on data item B. In the meantime, Application 2 locks data item B and then attempts to lock data item A. Neither can proceed.

This is the standard type of deadlock and can be avoided by an intelligent locking protocol (ILP). Under an ILP, all locks are acquired in the same order. In effect, an application must lock a row or record which acts as a *mutex* (an object which guarantees mutual exclusion). For instance, the application must obtain the lock on an order header before it can lock any of the detail rows or records for that order. In this way, if an application finds a data item locked, all it must do is wait until the owning application has released its locks; and it then can continue on. If an ILP is not being used, then deadlocks can be resolved by one or both applications timing out, releasing their locks, and trying again later at a random time.

Things are not so simple in a distributed system. Although all applications may be following a common ILP, consider an Application 1, running on node X, which locks data item A on its node X. Before that lock is propagated to node Y, Application 2 on node Y locks that same date item A on its node Y. Each application then attempts to lock the same data item on the remote node but

cannot. A deadlock has occurred even though both applications were following the same ILP.

This deadlock occurred because though both applications were following local ILPs for the nodes they were on, distributing the database and allowing lock access to all copies of all data items in any order on any node means that a global ILP needs to be used. However, there is a solution. One node of the many nodes in the application network must be designated the master node. Locks must first be acquired on the master node before attempting to acquire locks on other nodes. In this way, there is one and only one mutex for each lockable data set.

Failures and Recovery

Dual (Plural) Writes

If a node or network fails in a dual write environment, then the application will have to switch to single node operation. Upon restoration of the node or network, an on-line database comparison and update utility will have to be used to update the remote database (this sort of utility is not normally provided with systems; rather, the entire database must be copied unless a user-written utility is provided). As with asynchronous replication, if an isolated node continues in service, then data collisions may occur during the outage as well as during the recovery process. These collisions will have to be detected and resolved.

Coordinated Commits

Failover

When using coordinated commits, the recovery of user services following a node or network failure is almost identical to that for asynchronous replication, as described in Chapter 3, "*Asynchronous Replication*." In the event of a node failure, users may be switched from the failed node to the surviving node. If necessary, full operation

resumes with the load shedding of non-critical functions. There is one important difference.

With asynchronous replication, transactions in the replication pipeline from the failed node are lost. With synchronous replication, no data is lost. Rather, one or more transactions may be held in an uncertain state. That is, if a node owning an outstanding transaction should fail after its target nodes have acknowledged its prepare-to-commit command or its RTC token but before the target nodes have received the commit command, then the target nodes do not know whether the source commanded either a commit or an abort or whether the source failed before it could do either. In this case, the target transactions are hung but are not lost. What is done with these transactions is application dependent, but at least the data is not lost.

Should the failure be a network failure, then it may be decided to switch users to a node which is not isolated. An alternative decision may be to allow the isolated node to continue independently.

Restoration

In any event, during the outage, transaction updates will queue at the active nodes and will be sent later to the downed or isolated node when it is returned to service. At this time, replication proceeds as asynchronous replication to bring the remote database into synchronization. If isolated nodes continue in service so that transactions are being replicated in both directions, collisions will have to be detected and resolved as described in Chapter 3.

When the Change Queues have been drained to the point that the replication latency is deemed to be acceptable, users can be re-switched to their home nodes; and synchronous replication can be restarted. Synchronous replication messages will simply queue behind the remaining asynchronous messages until those asynchronous

messages have been processed, at which time the application returns to purely synchronous replication.

What's Next?

In Chapters 1 and 2, we viewed availability with respect to systems which require repair and which are returned to service the instant the repair is complete. However, in the real world of redundant systems, a system seldom needs repair to return it to service. Rather, faults are transient; and the system needs to be recovered rather than repaired. Chapter 5, "*The Facts of Life*," discusses this twist on system availability.

There are three major categories of failures with that to be concerned relative to asynchronous replication:

- the source node fails.
- the network fails.
- the target node fails.

Of course, in a bi-directional replication environment, a single-node failure represents both a source node failure and a target node failure because the failed node had been performing both functions.

There are two recovery points of interest in a data replication environment. One is the immediate recovery of processing functions following a failure. We will call this *failover.* The other is the return to service of the failed component once it is repaired. We will call this *restoration.*