"Breaking the Four 9s Barrier"
An Informational Series on Enterprise Computing

As Seen in *The Connection*, An ITUG Publication
September 2002 – December 2003

About the Authors:

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.

Series Topics:

Breaking the Four 9s Barrier, Part 6 - RPO and RTO (12/03)
Breaking the Four 9s Barrier, Part 5 - The Ultimate Architecture (9/03)
Breaking the Four 9s Barrier, Part 4 - Facts of Life (6/03)
Breaking the Four 9s Barrier, Part 3 - Sync Replication (4/03)
Breaking the Four 9s Barrier, Part 2 - System Splitting (2/03)
Breaking the Four 9s Barrier, Part 1 - The 9s Game (11/02)
Breaking the Four 9s Barrier, Part 0 - Intro/About the Authors (9/02)

***Gravic, Inc*.**
Shadowbase Products Group
17 General Warren Blvd.
Malvern, PA 19355
610-647-6250
www.ShadowbaseSoftware.com

Availability (Part 3) – Synchronous Replication

Dr. Bill Highleyman
Paul J. Holenstein
Dr. Bruce D. Holenstein

In "Part 2 – System Splitting"[1] of this series on availability, we looked at the availability advantages of replicating or splitting a processing system. *Availability* is the probability that a system will be operational. We measure availability in terms of 9s. For instance, an availability of three 9s means that the system will be up 99.9% of the time.

Replicating Systems

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 all of the transactional load. As shown in Figure 1a, updates made by the primary system to its data base are replicated to the backup system. The backup system is passive except for perhaps supporting read-only operations such as query and reporting.

Fully replicating a system, in addition to providing protection from disasters, has the effect of doubling its 9s, dramatically improving its availability.

Splitting Systems

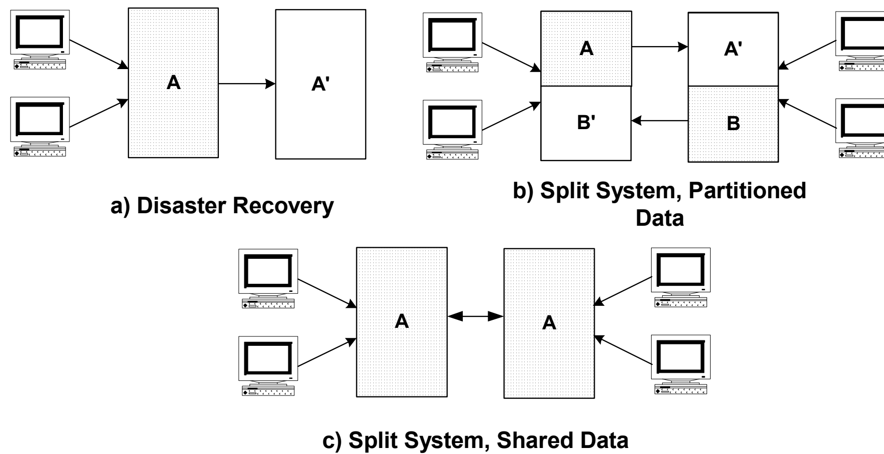
Alternatively, significant availability advantages can be achieved by simply splitting a single system into *k* nodes. In Part 2, we showed that doing so could increase system reliability (i.e., increase its mean time before failure) by more than a factor of *k*.

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 data base.

Often, the data base 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 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 system to the target system. Updates in the replication pipeline may be lost in the event of a system failure.

[1] Highleyman, W., Holenstein, B. "Availability Part 2 – System Splitting," The Connection, Vol. 24, No. 1; January/February 2003.

However, in the more general case, any system in the network must be able to update any data item (Figure 1c). Those updates then must be replicated to the other data bases 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 simultaneously. These conflicting updates then would be replicated across the network, resulting in a corrupted data base (i.e., the value of the data item would be different in different instances of the data base). We call these simultaneous conflicting updates *data collisions.*



Split System Architectures
Figure 1

Data Collisions

The probability that data collisions will occur is surprisingly high. Jim Gray has shown[2] that the data collision rate in such a network is given by

$$\text{Data Collision Rate} = \frac{td(ruk)^2}{2D}$$

where

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

[2] 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.

Consider a system split into four nodes with a requirement to maintain an up-to-date data base 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 (u=4) and requires 200 milliseconds to complete (t=.2). Furthermore, let us assume that our data base requires 10 gigabytes and that an average row (the lockable entity) takes 1000 bytes. Thus, the data base contains 10 million lockable objects (D=10,000,000).

Using the above relation, we find that this system will create almost four collisions per hour. This can be a major headache. If the nodal transaction rate increases to 100 transactions per second, then the collision rate jumps to almost 400 collisions per hour. This would certainly keep a team of people busy. If the system then grew to eight nodes, the collision rate would explode to almost 3,000 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 data base is often a lengthy and manual operation since automated resolution rules are often not practical.

Synchronous Replication

This Part 3 of our availability series deals with methods for avoiding data collisions in active/active applications rather than having to correct them.[3] 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.[4]

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 they 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 system may be held up until all data items across the network have been updated. In this paper 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. The analysis is easily extended to systems comprising multiple nodes, and the conclusions would be the same.

---

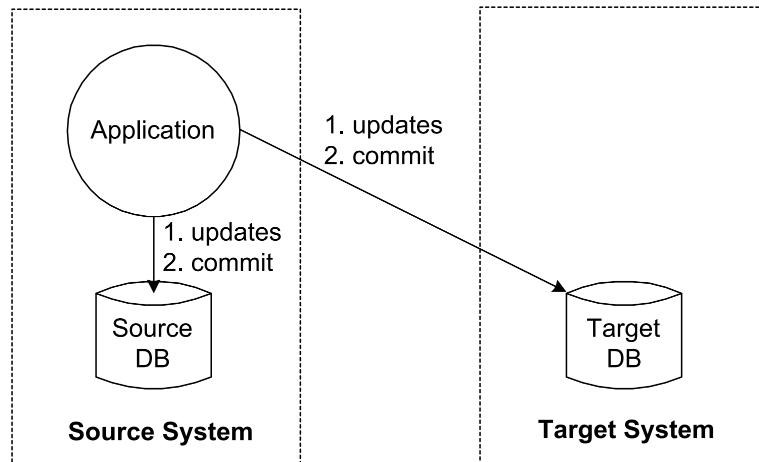
[3] See United States Patent Application No. 2002/0133507, "Collision Avoidance in Data Replication Systems," Holenstein et al; Sept. 19, 2002.

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

Dual Writes

In the NonStop world, the term "dual writes" is just another name for network TMF. Dual writes involve the application of updates within a single transaction to all replicates of the data base. 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 data bases will always have the same value and that the data bases therefore are always consistent.

A simple view of synchronous replication using dual writes under a transaction manager is shown in Figure 2. A transaction is started by the application, and updates are made to the source data base and also to the target data base across the network (1). 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 data bases (2). If the transaction manager is unsuccessful in doing this, then the transaction is aborted; and no updates are made.



Dual Writes
Figure 2

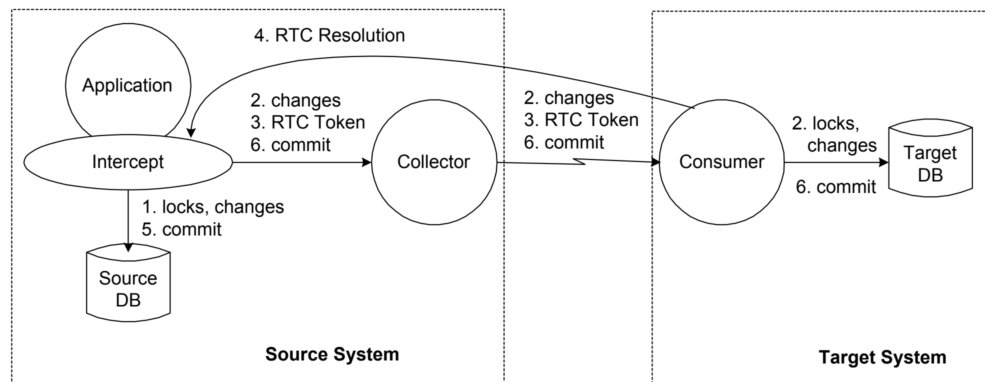
Note that transactional updates under a transaction manager may be done either serially or in parallel. If the updates are serial, then the application must wait not only for the local updates to be made but also for the update of each remote data item over the network. If updates are done in parallel, then to a first approximation the application is delayed only by the communication channel propagation time. The read/write time at the target system is transparent to it.

Coordinated Commits

An alternative approach to dual writes is to begin independent transactions on each system and then to *coordinate the commits* of those transactions so that either they both commit or that neither commits. In this case, the updates may be propagated asynchronously to the target system using normal data replication techniques. There they are applied directly to the target system's data base as part of its transaction that was started on behalf of the source system. 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 3. The application first will start a local transaction. As the application locks data items and makes updates to its source data base (1), the changes to the data base are captured (either by reading an audit file or log file or by intercepting the update commands from the application). These updates are sent to the target system (2), where a transaction is started, locks are acquired, and the updates are made to the target data base.

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



Coordinated Commits
Figure 3

Should the target system respond negatively to the RTC Token, then the transaction is aborted at the source system. There is the possibility that the source commit might succeed and the subsequent target commit fail. Since the target system 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 system or the network should fail after the target system has returned the RTC Resolution and before it receives the commit directive from the source system.

In this case, the normal data resynchronization capabilities of the replication engine would have to be invoked to resynchronize the data bases. However, the data bases would be out of sync anyway when the target system is returned to service. The lost transaction is just one of many that will be recovered through the resynchronization procedure.

Resynchronization

For active/active applications, both the dual write method and the coordinated commit method for synchronous replication must face the possibility that the data bases may get out of synchronization because of a system or network failure. In such cases, the applications generally will continue on the operational systems; but data updates will not be replicated. When the fault is corrected, the data bases must be resynchronized; and any data collisions must be resolved.

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 system. We call this additional delay the *application latency* caused by synchronous replication.

Note that if data replication were asynchronous (as is the usual case) rather than synchronous, then the application would not be delayed by data replication. Instead, the target data base would lag behind the source data base 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 local operations. 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, inserts and replacements that require only one access of the data base. It is quite straightforward to modify the following relationships to account for these factors. If we did this, the general conclusions that we would reach would be unchanged.

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

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

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

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

In Equation (1), the first term is the communication time required to send the updates to the target system. 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.[5]

Coordinated Commits

Coordinated commit replication requires the use of a data replication facility to propagate the updates to the target system. Most of the time spent by this facility is invisible to the application providing that updates to the remote data base 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 system, the subsequent commit at the target system is asynchronous relative to the application. Its success is guaranteed since the

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

target system 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 system. Therefore, the target system commit 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 it is received by the target system 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,

$$L_{cc} = t_p + 2t_c \tag{2}$$

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 (1) and (2),

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

Equation (3) can be rewritten as

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

where

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

$$p = 2t_c / t_p \quad (5)$$

Figure 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 is shown.

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

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

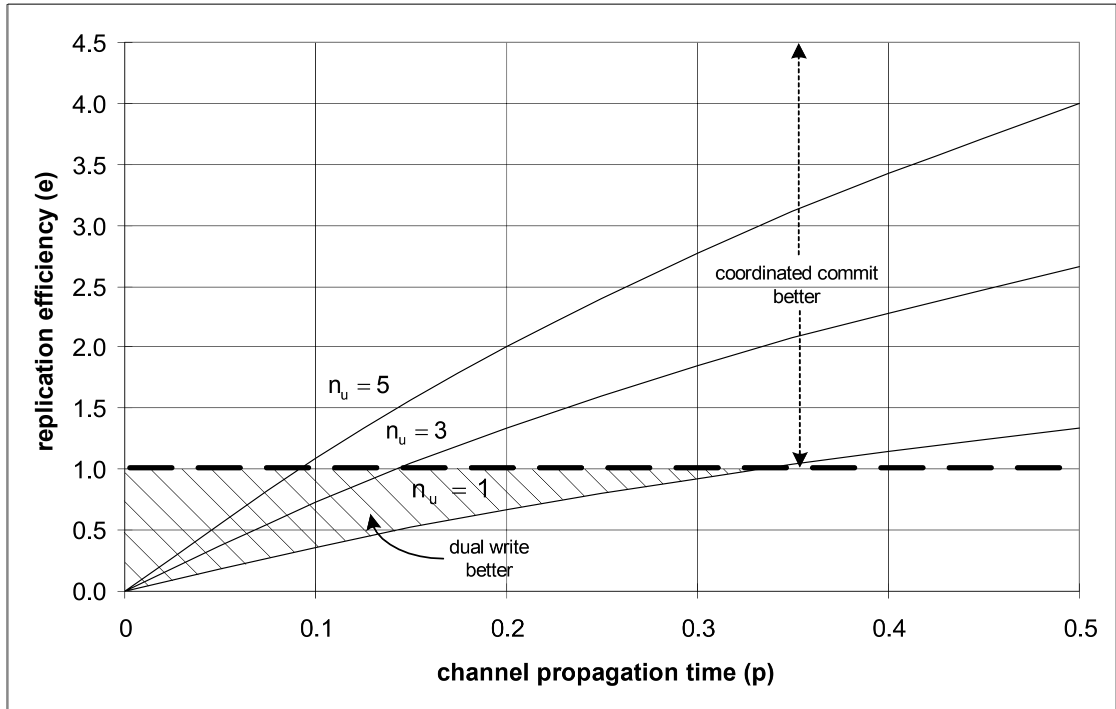
or

$$p = 1/(2n_u + 1) \quad (6)$$

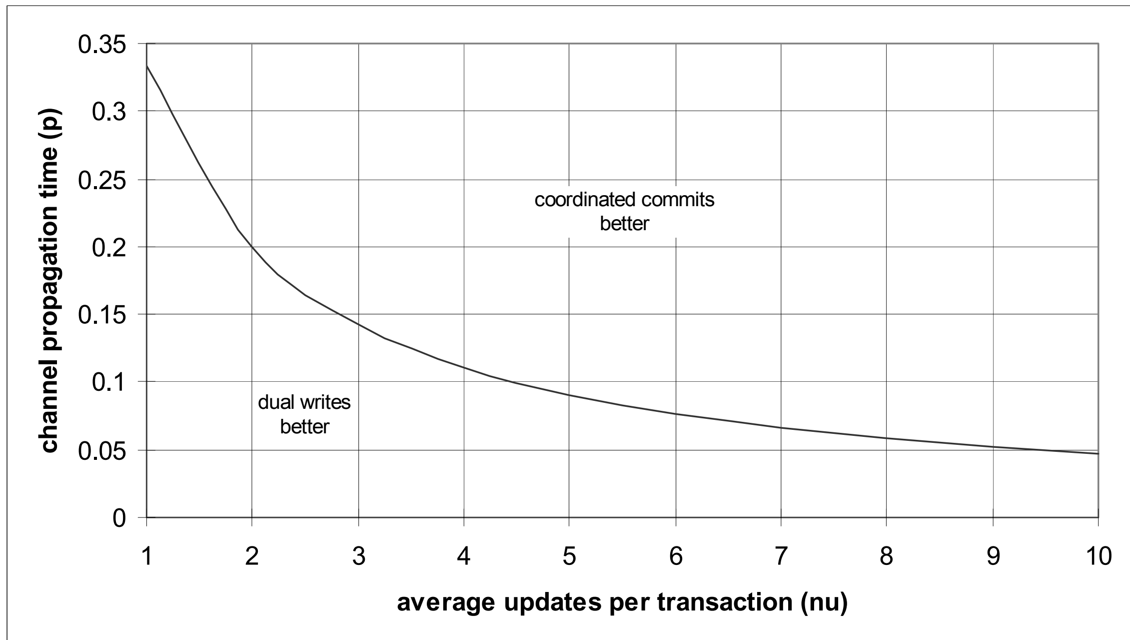
This relationship is shown in Figure 5.

Figures 4 and 5 are for the case of parallel read/writes. 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 (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. These curves may be modified easily to reflect this situation.



Synchronous Replication Efficiency
Figure 4



Equal Efficiency (e=1)
Figure 5

Scalability and Other Issues

There are other issues to consider when comparing synchronous{ XE "Synchronous" } replication algorithms, including scalability (as it relates to performance) as well as various other algorithm optimizations.{ XE "Scalability" }{ XE "Data replication: highly scalable" } We now discuss these issues, including the effects of multiple database copies, communication channel efficiency{ XE "Communication channel: efficiency" }{ XE "Data replication: synchronous: efficiency" }, and transaction profile.

Multiple Database Copies

If there are *d* database copies in the application network, then an application using dual writes{ XE "Dual writes" }{ XE "Data replication: synchronous: dual writes" } must make *d* modifications for every database modification required by the transaction. However, under coordinated commits{ XE "Data replication: synchronous: coordinated commits" }{ XE "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{ XE "Data replication: engine" } and do not affect the application.

Therefore, if other processing is ignored, application latency{ XE "Latency: application" }{ XE "Data replication: synchronous: application latency" } (which adds to transaction response time) under dual writes{ XE "Dual writes" }{ XE "Data replication: synchronous: dual writes" } will increase with the number of database copies, *d*, whereas the transaction response time for coordinated commits{ XE "Data replication: synchronous: coordinated commits" }{ XE "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{ XE "Dual writes" }{ XE "Data replication: synchronous: 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{ XE "Data replication: asynchronous: replication latency" }{ XE "Data replication: asynchronous: 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{ XE "Dual writes" }{ XE "Data replication: synchronous: dual writes" }. However, the number of messages sent by coordinated commits{ XE "Data replication: synchronous: coordinated commits" }{ XE "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{ XE "Dual writes" }{ XE "Data replication: synchronous: dual writes" } will grow proportionately. However, network traffic generated by coordinated commits{ XE "Data replication: synchronous: coordinated commits" }{ XE "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{ XE "Dual writes" }{ XE "Data replication: synchronous: dual writes" } and coordinated commits{ XE "Data replication: synchronous: coordinated commits" }{ XE "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{ XE "Data replication: synchronous: coordinated commits" }{ XE "Coordinated commits" }, the source application transaction runs at full speed until commit time, and then pauses while the RTC{ XE "Coordinated commits: ready-to-commit (RTC) token" } 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{ XE "Latency: application" }{ XE "Data replication: synchronous: 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{ XE "Dual writes" }{ XE "Data replication: synchronous: 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{ XE "Intelligent locking protocol (ILP)" }{ XE "Deadlocks:

intelligent locking protocol (ILP)" } 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{ XE "Data replication: synchronous: coordinated commits" }{ XE "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{ XE "Data replication: synchronous: 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	150 msec.	replication engine processing time, or the time to propagate an update through the replication engine from the source system to the target system.
t_c	25 msec.	communication channel propagation time, or the amount of time required for a message to propagate from the source system to the target system or vice versa over the provided communication channel, including line driver and transmission time.

These parameters result in a value for p (from Equation (5)) of .33. The resulting efficiency for a given transaction size can be read from the graphs of Figure 4. Alternatively, from Equation (3),

$$e = \frac{50(n_u + 1)}{100}$$

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

n_u	e
1	1.0
2	1.5
3	2.0
4	2.5

As is seen, coordinated commits are more efficient for all transaction sizes greater than one update in this example. This is a direct result of the communication channel delays. In this example, the application latency due to synchronous replication is, from Equation (2), 200 msec.

Generally, dual writes may be more efficient when the communication times are short compared to disk times. Note 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 could take about 250 msec.

Consequently, a very generalized statement is that dual writes are appropriate for campus environments with small transaction sizes. However, 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 gotten out of sync due to network or node failures, a capability which would probably have to be specially developed for dual writes.

In general, as can be seen from Figures (4) and (5) and Equation (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 Parts 1 and 2, we have

Rule 10: *For synchronous replication, coordinated commits using data replication become more efficient relative to dual writes under a transaction manager 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 will generally 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 system, 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. This would be a typical ServerNet case. In this case, *p* is .027 and efficiency from Equation (3) is

$$e = \frac{4(n_u + 1)}{77}$$

Solving this for *e = 1*, we find that replication for transactions exceeding 18 updates will be faster for coordinated commits than it will be for dual writes. For an 18-update transaction, the application latency is, from Equation (1), 152 msec.

What's Next

In Parts 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 NonStop 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. "Availability (Part 4) – The Facts of Life" discusses this twist on system availability.

In Part 5, we take the concepts which have been developed in the first four parts and suggest a system architecture that can dramatically increase system availability without a significant cost increase.