

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.

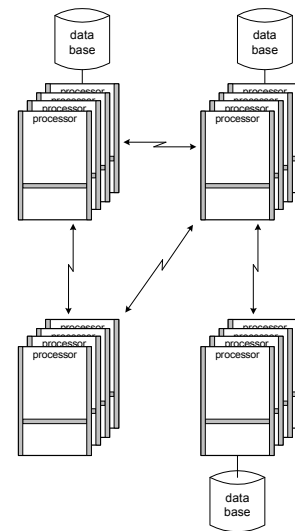
Chapter 3 - Asynchronous Replication

In Chapter 2, “*System Splitting*,” we showed that distributing an application over several independent nodes can significantly increase the availability of the overall system. Of course, in order for the application to be available, it must not only have processing resources, but it must also have access to the application data. Therefore, in order to ensure application availability, there must be a copy of the application database at at least two of the nodes; and all processing nodes must have access to all copies of the database, as shown in Figure 3-1.

Not all nodes need to have a local copy of the database, and not all nodes need to be directly connected to each other. What is required is that each node have a path to all copies of the database, wherever they may be, so that the results of each transaction can be reflected in all database copies. That is, the various copies of the databases need to be kept in the same state so that the use of any database copy by any node will provide the same result.

Therefore, any change made to one database copy must be immediately propagated to all of the other database copies. We call this *data replication*, and its purpose is to keep synchronized the various copies of a database in a redundant system.

There are several ways to do data replication – asynchronously, synchronously, physically, and others. In this chapter, we discuss



Distributed Application
Figure 3-1

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

asynchronous replication, and we discuss synchronous replication in the next chapter.

Uses for Data Replication

There are many advantages to distributing an application across multiple independent nodes:

Disaster Tolerance

The processing nodes may be geographically distributed. In this case, should one node be disabled or destroyed by a natural or man-made disaster, the application will survive and continue to function with no data loss, though perhaps in a degraded mode.

Increased Availability

Even if the nodes are not geographically distributed, the overall system availability can be significantly enhanced by breaking up the system into multiple smaller nodes with no increase in processing or storage resources. This is directly due to a reduction in the number of failure modes for the system. For instance, as shown in Chapter 2, if a large monolithic system is separated into k independent nodes, then the total system availability is increased by at least a factor of k . Furthermore, should there be a failure of a node, only $1/k$ of the system capacity is lost. The chance of losing more than $1/k$ of the total system capacity is virtually never.

Localization

If the users of an application are widely distributed geographically, then processing and data storage resources can be provided much closer to each user, thus improving application response time. In fact, each user community can be served by a node whose size and configuration are more closely attuned to the needs of that user community. In addition, the vendor hardware, operating system, and database product at each node can be chosen

independently to minimize costs, to improve manageability, or to otherwise optimize the node for its particular user community.

System Maintenance

If a monolithic system needs to be updated or otherwise maintained, it often must be brought down, thus contributing to undesirable scheduled down time. If the system is distributed, then system maintenance can be applied to one node at a time. During this activity, only one node in the system needs to be down. All other nodes continue to provide full service to the user community.

Enterprise Application Integration (EAI)

Many enterprises are now integrating their disparate applications running on systems from different vendors into a common network for increased functionality, efficiency, and customer service. Many use middleware products to do this. However, this approach generally means that the applications have to be modified to interface properly with the middleware facility. Heterogeneous data replication can provide up-to-date copies of commonly used data on disparate databases in the format expected by each database with no application modifications required.

Operational Data Store (ODS)

One powerful technique for EAI is to implement an independent operational data store that is central to all systems in the enterprise. Any data with immediate value to the enterprise is replicated instantly to the ODS. Data in the ODS is available to all applications no matter where they may reside, thus giving these applications a current view of all data within the enterprise.

Data Warehousing

In a manner similar to supporting an ODS, data replication can be used to feed a data warehouse. Many data replication engines allow the data to be extensively manipulated before applying it to the target database, a requirement for data warehouses that typically archive summary results only.

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

Zero Down Time System Migration

Data replication can be a powerful way to migrate an application from one system or database to another with zero down time. In principal, the new system is synchronized with the old system while users continue normal activity on the old system. When the new system is synchronized, the users may be switched to the new system. If desired, the new system may continue to replicate its changes to the old system so that a return can be made to that system should the new system experience problems.

Types of Data Replication

In general, a replicated system will have multiple nodes as shown in Figure 3-1, and data replication will be ongoing between all nodes that contain part or all of the database. However, each of these replication paths is commonly served by an independent data replication engine. It is the concurrent operation of all of these replication engines that maintains synchronism between the databases that are distributed across the application network.

Therefore, it suffices to talk only about a single replication engine. A distributed application comprises two or more processing nodes using one or more replication engines to keep its database copies synchronized.

There are many ways to implement a data replication engine, and each has its strengths and weaknesses. We consider in this section five important replication engine characteristics:

- directionality
- threading
- queuing
- target updating
- synchronism

Directionality

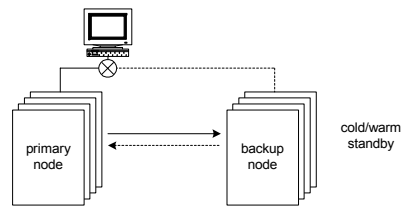
By directionality, we mean whether data replication between two nodes is done in only one direction (unidirectional) or in both directions (bi-directional active/active); and if bi-directional, can a particular data item be updated by only one node (partitioned active/active) or by any node (non-partitioned active/active). These various cases are shown in Figure 3-2:¹⁹

- **Unidirectional:** Unidirectional data replication is typically provided for disaster recovery, data warehousing, and operational data stores. These configurations generally comprise a primary node that is providing all processing functions and a backup node that is ready to take over these functions should the primary node fail, as shown in Figure 3-2a. A current copy of the database is maintained at the backup node by replicating all changes made to the database at the primary node to the backup node. For fast recovery, the backup node may also be configured for replication to the primary node; but this path is inactive unless the backup takes over.
- **Bi-Directional Partitioned Active/Active:** A more efficient use of a replicated system, known as *active/active* or *cooperative processing*, is for all nodes to be cooperating in the processing of the application's transactions. All nodes replicate their changes to all database copies at other nodes. One problem with cooperative processing occurs if two users at different nodes attempt to modify the same data item at the

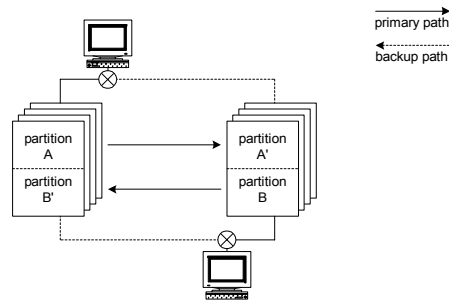
¹⁹ Standish Group calls these architectures Functional Segmentation (Unidirectional Replication), Application/Database Segmentation (Bi-Directional Partitioned Active/Active Replication), and Live/Live (Bi-Directional Non-Partitioned Active/Active Replication). In a poll of high-availability users, Standish Group found that 81% favored the active/active approach. See Standish Group Research Note, "The New High Availability – A NonStop Continuous Processing Architecture (CPA);" 2002.

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

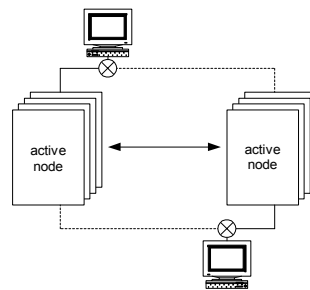
same time. This problem is analyzed later, but it is avoided if the database can be partitioned so that each data item is in effect *owned* by one and only one node. Only the owner of a data item can update that data item, thus avoiding conflicts. This is the case of partitioned data replication, as shown in Figure 3-2b.



a) Unidirectional Replication



b) Bi-Directional Partitioned Active/Active Replication



c) Bi-Directional Non-Partitioned Active/Active Replication

Replication Directionality
Figure 3-2

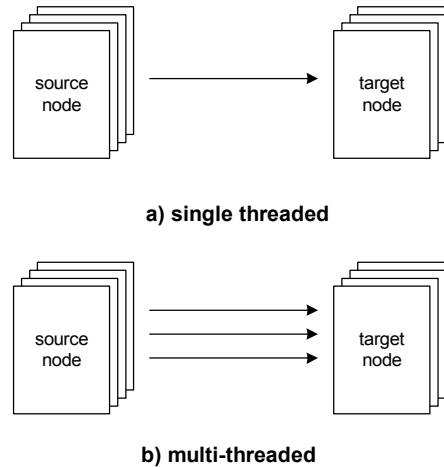
- **Bi-Directional Non-Partitioned Active/Active:** The most general case for cooperative processing is to allow any user at any node to modify any data item in the network, as shown in Figure 3-2c. This is the most flexible and powerful case of cooperative processing, and it is known as active/active data replication. However, in this case, data access conflicts may occur, sometimes with alarming frequency. Means must be provided to resolve data conflicts. Some are described later in this chapter (for data collisions) and others in Chapter 4, “*Synchronous Replication*,” (for deadlocks). The potential frequency for data conflicts is analyzed in some detail in Chapter 9, “*Data Conflict Rates*.”

Threading

In some applications, the peak rates for data replication may overwhelm a simple data replication engine. In this case, multiple replication threads may be used, as shown in Figure 3-3.

Each thread represents an independent flow of database modifications from the source database to the target database, thus substantially increasing the capacity of the database engine. However, since modification event flow over each thread is independent of all other threads (absent any inter-thread coordination), there is no guarantee that any inherent order in these events will be maintained at the target node.

It may be (and usually is) imperative that some means be provided to ensure a certain order in the applying of these modifications to the target database in order to prevent data corruption or inconsistency. This problem is discussed briefly later in this chapter and is extensively analyzed in Chapter 10, “*Referential Integrity*.”



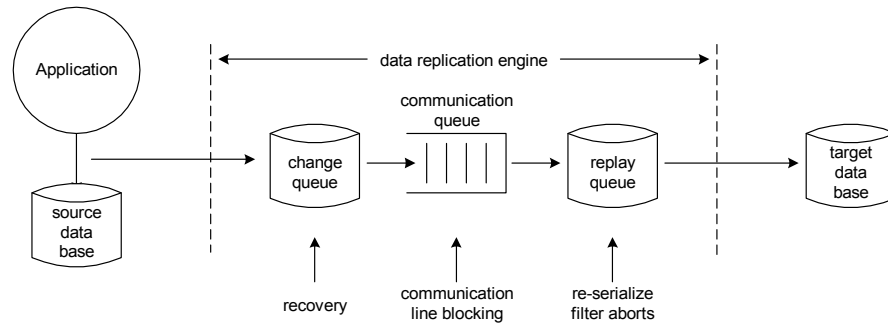
Data Replication Threads
Figure 3-3

Queuing

There are many implementations of replication engines. Some are designed to be very fast in terms of getting source database changes applied to the target database. Others queue changes at various points throughout the replication engine for a variety of reasons, as shown in Figure 3-4:

- to minimize the amount of lost data in the event of a network or node failure.
- to provide for proper serialization of changes to the target database.
- to control the applying of changes that are provided by non-recoverable triggers at the source node.
- to prevent transactions that are ultimately aborted from being applied to the target database.

- to improve efficiency of certain components (e.g., blocking messages over a communication channel).



Data Replication Engine Queuing
Figure 3-4

One form of queuing is typically always required, and that is to queue source database changes on a persistent store such as disk at the source node so that they can be replayed following a network or target node failure. True, the persistent store does not need to be used during normal operation; but it must be put into play in the event of a network or target node failure in order to save database changes for later replication.

A replication engine with no queuing points represents the fastest architecture. To the extent that there are one or more queuing points, the interval between the time that a change is applied to the source database and the time that it is applied to the target database is lengthened. This interval is called the *replication latency*. Replication latency affects data replication in many ways and is described later in this chapter as well as in other chapters.

Target Updating

There are several algorithms for how changes are applied to the target database. Each has its own merits in certain applications.

- **Absolute Replication:** When using absolute replication, the data replication engine sends the after images to the target database; and these are used to overwrite the current data

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

items. After images may represent an entire row or record or just the changed data within a row or record. (For deletes, only the row or record key needs to be sent.)

- **Relative Replication:** Alternatively, only the relative differences or change operations are replicated. These differences may be determined by comparing the before and after images of the changed data at either the source node or target node. The relative difference is, in effect, an operation on the target data. For instance, if a source data field is changed from 10 to 15, the replicated relative value will be “add 5” rather than “replace with 15” as is the case for absolute replication.
- **Fuzzy Replication:** Fuzzy replication can be advantageous if the databases are not kept in exact synchronization. If certain replication operations fail, then they are mapped to other operations and retried. For example:
 - if an insert fails because the record already exists, an update is done instead.
 - if an update fails because the record does not exist, an insert is done.
 - if a delete fails because the record does not exist, the operation is ignored; or perhaps a row or record is reinserted with the relative difference.

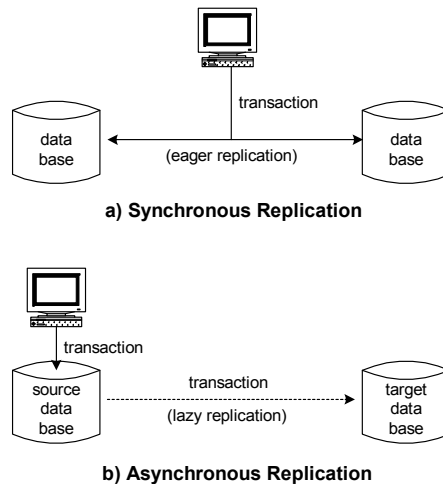
Synchronism

We have characterized data replication as a means to keep distributed databases synchronized. A more accurate characterization might be to say that data replication will keep database copies exactly synchronized or nearly synchronized. The degree of synchronization depends upon the characteristics of the replication engine.

Specifically, database replication techniques include synchronous replication, that will keep the database copies exactly synchronized,

and asynchronous replication, that will keep the database copies nearly synchronized. Each has its benefits and problems.

- **Synchronous Replication** is shown simply in Figure 3-5a. In the simplest of terms, all changes to all copies of the database are treated as a single transaction. Either all are made, or none are made. More specifically, the replication engine will obtain locks on all data items in all database copies. It will then make all changes before releasing the locks (i.e., the transaction is committed). If it cannot make all changes, it will restore the original values of the data items before it releases its locks (i.e., the transaction is aborted). In this way, the view of any database copy at any point in time is the same as the view of any other copy. The databases are kept in exact synchronization. Gray²⁰ calls this *eager replication*.



Replication Synchronism
Figure 3-5

Clearly, by behaving in this manner, the application is

²⁰ 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.

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

somewhat delayed because it must coordinate the modification of two or more database copies across the network. The additional time that it takes a transaction to complete in this distributed environment as compared to the time that it takes in a system with a single database copy is called *application latency*.

- **Asynchronous Replication** is shown in Figure 3-5b. In this case, the application is not held up by the data replication process. Rather, the application operates strictly on a single database instance. Behind the scenes, the data replication engine is monitoring changes to the source database and is sending these changes to be applied to the target database(s). The application is totally unaware of this activity.

In the above case, there is no application latency. However, there is a time delay from the time that a change is made to the source database and the time that the change is applied to the target database. This time delay is known as *replication latency* and was described earlier with respect to replication engine queuing. Thus, the target database lags the source database by the replication latency time; and the database copies are *nearly* synchronized. Gray refers to this as *lazy replication*.

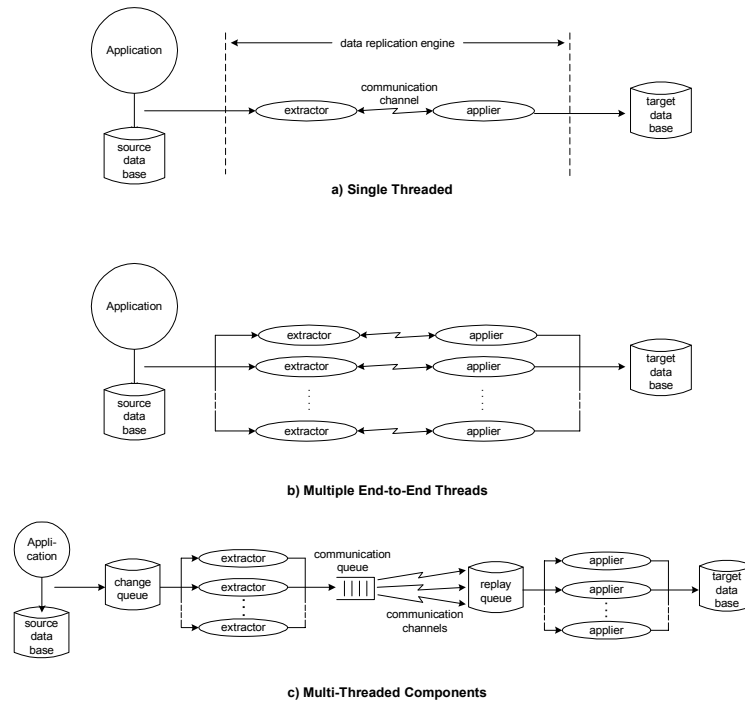
The rest of this chapter deals with issues relating to asynchronous replication. Synchronous replication and application latency are discussed in Chapter 4, “*Synchronous Replication*.”

The Basic Data Replication Engine

Before discussing some of the issues with respect to asynchronous replication, let us look at the basic architecture of a data replication engine. Architectures for replication engines are described in much more detail in Chapter 10, “*Referential Integrity*.”

A replication engine generally has three main components, as shown in Figure 3-6a:

- an Extractor on the source node. The Extractor monitors the source database for changes.
- a communication channel that the Extractor uses to send changes to the target node.
- an Applier on the target node. The Applier receives changes from the Extractor and applies them to the target database.



Data Replication Engine Architectures
Figure 3-6

Although Figure 3-6a shows the Extractor on the source node and the Applier on the target node, this is unnecessary. The Applier can also be on the source node (and perhaps be in the same process as the Extractor) and will “push” database changes to the target database via remote procedure calls (RPCs) or some similar mechanism. Likewise,

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

the Extractor can reside on the target node, can be part of the Applier, and will “pull” database changes over the communication channel.

Figure 3-6a illustrates a single-threaded data replication engine. Alternatively, for performance purposes, the replication engine can be multi-threaded. Figure 3-6b shows multiple end-to-end threads that can operate independently to carry their own stream of database changes. Not shown is a mechanism for re-serializing these changes into proper order if needed before applying them to the target database.

Figure 3-6c shows a more complete picture of a different type of multithreading. It shows that each of the replication engine components – the Extractors, communication channels, and Appliers – can be individually multithreaded. Any combination of component single-threading and multithreading can be provided.

In addition, major optional queuing points are shown in Figure 3-6c. They are only implied in Figures 3-6a and 3-6b:

- **Change Queue:** Changes extracted from the source database are commonly written to a persistent store such as disk so that they can be replayed in the event of a network or target node failure. During peak transaction rates, this queue also helps buffer changes that might otherwise overwhelm the replication engine. The Change Queue can be found in many forms, including:
 - an Audit Trail maintained by the node’s transaction manager.
 - a Change Log created by the application.
 - a Database of Change (DOC) file created by the replication engine if there is no external Change Queue created outside of the replication engine.
- **Communication Queue:** Changes are often held at the communication channel and accumulated so that large blocks

of changes may be sent over the communication channel. In this way, the replication engine sends an occasional (relatively speaking) large message rather than many frequent small messages. This will, in many cases, significantly increase the efficiency of use of the communication channel and will reduce communication delays.

- **Replay Queue:** If multithreading is used, re-serialization of changes arriving at the target node may be required to ensure that changes are applied to the target database in the correct order. One way to achieve this is by storing all changes as they arrive in a Replay Queue (that may be disk-resident). Changes are then read from the Replay Queue in correct order using one or more closely coordinated Appliers, that will apply the changes to the target database node in the correct order.

The Replay Queue or the Change Queue can also be used to store an entire transaction before applying it or sending it to the target node. In this way, changes that are part of transactions that are later aborted need not be applied or sent to the target database. This is especially useful if the target database does not support the notion of transactions, that means that a partially applied set of changes cannot be backed out or aborted.

Advantages of Asynchronous Replication

There are several important and positive characteristics that are typical of these asynchronous replication architectures.

No Performance Penalty

The addition of an asynchronous replication engine to an application generally imposes little application performance penalty since all replication activity is independent of the application. Replication is totally decoupled from the applications and interacts with them only via the Change Queue.

Non-Invasive

The addition of a data replication engine to an existing application is often non-invasive. It requires no changes to the application. The data replication engine acquires source database changes through mechanisms that often already exist (such as an Audit Trail). Therefore, it is easy to retrofit existing applications for disaster recovery or for EAI or to distribute them to improve availability or locality.

Heterogeneous

Replication may be between heterogeneous databases or even between heterogeneous systems since the Extractors and Appliers are loosely coupled through a messaging system (the communication channel) and can be tailored to their own environments. For instance, the source node can be a legacy system using flat files; and the target node can be an open system using a relational database.

Data Manipulation

The replication engine can provide virtually any data manipulation function of the data prior to applying it to the target database. This can include simple functions such as data format conversion or complex functions such as table look-up, aggregation, and computation. Data manipulation is application dependent. It may be supported by scripts or user exits usually made available within the source or target node replication engine components such as the Extractor or the Applier.

Data Integrity

The replication engine will ideally apply each event once and only once to the target database. The replication engine can be configured so that the applying of modifications and transactions is in the same order as that at the source database. The preservation of the *natural flow* of database modification events guarantees that the target database will always be an exact copy of the source database, though perhaps somewhat delayed by the replication latency.

Highly Reliable

The replication engine can be designed to be highly reliable in that a source node will never lose any changes as long as they are stored in a persistent store. In this way, the full recovery of a downed or isolated node now restored to service can always be guaranteed by replaying changes that it had not received.

Highly Available

The replication engine can be designed to be highly available, especially if running in a fault-tolerant environment. Extractors and Appliers can be implemented as primary/backup process pairs running in different processors. Alternatively, they can be configured as persistent processes that will be restarted (in a different processor if need be) following a failure. This function is usually provided by a fault-tolerant monitor process pair or as an operating system service.

Highly Scalable

The replication engine is inherently highly scalable. Since each replication engine handles just one source/target database instance, adding database instances to the application network does not increase the load on any one replication engine. Rather, additional replication engines are simply added for each additional node pair. For instance, a two-node system will have two replication engines, one for each direction. A four-node system will have at most twelve replication engines, one pair for each of the six possible connections.

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

Highly Secure

The replication engine is decoupled from the source applications and the source database and so does not provide a path into the source node for unauthorized access. Furthermore, the specific data to be shared with other nodes can be tightly controlled.

Asynchronous Replication Issues

Discussed above are some of the many advantages of asynchronous replication. However, there are also many issues with that to be concerned. All can become non-issues if handled correctly.

General Issues

Some of these issues are independent of the type of asynchronous replication. They include the consequences of replication latency and multithreading.

Data Loss

The fact that there is replication latency associated with asynchronous replication means that at any one time there are transactions in the replication pipeline flowing from the source node to the target node. Should a node fail, the transactions that it had in the pipeline are lost. True, they can be recovered from the source node's Change Queue once that node has recovered. While the node is down, however, the transactions will be unavailable.

The amount of data that may be lost is directly proportional to replication latency. Minimizing replication latency will minimize the potential for data loss in the event of a source node failure.

Rule 11: *Minimize data replication latency to minimize data loss following a node failure.*

However, if this temporary loss of transactions is considered a serious and perhaps unrecoverable problem, then synchronous

replication as described in the next chapter is a good solution. Synchronous replication guarantees that no data is ever lost.

Database Corruption

It is often very important to apply changes to the target database in the same order as they were applied to the source database. This is called the *natural flow* of changes and transactions. If natural flow is violated, there are several opportunities for database corruption.

For instance, if two absolute updates to the same row or record are applied in reverse order, the data item will be left at the older value. Or a child record may be inserted before a parent record, causing a referential integrity violation. This may cause the target database to reject the child insert.

Rule 12: *Database changes generally must be applied to the target database in natural flow order to prevent database corruption.*

A single-threaded replication engine as shown in Figure 3-6a will typically guarantee that the event order delivered to the target database is identical to the event order received from the source database. However, if the replication engine is multithreaded, as shown in Figures 3-6b and 3-6c, there is no guarantee of order at the target database since changes and transactions may flow over the different threads at different rates. Therefore, a mechanism for re-serialization must be provided following all multithreaded activity and prior to the target database. This is exemplified by the Replay Queue shown in Figure 3-6c.

Interestingly, multithreading is employed to increase the capacity of a data replication engine. However, even though the capacity of the replication channel may increase when multithreading is used, the requirement for re-serialization may increase the actual replication latency depending upon the re-serialization method employed.

One technique to avoid database corruption due to improper modification event order is to hold up the replication of a transaction

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

until all previous transactions have been committed. However, this might not only slow down the replication channel but also may cause peak loads that were not observed on the source database to occur on the target database.

Rule 13: *Follow natural flow order when replicating so as not to create artificial activity peaks at the target database.*

The effects of multithreading and techniques for ensuring that multithreading does not cause data corruption are extensively analyzed in Chapter 10, “*Referential Integrity*.”

Ping-Ponging

Ping-ponging is an issue when bi-directional replication is used. If care is not taken, a change replicated from a source node to a target node can be replicated back to the source node, then back to the target node, and so on ad infinitum. This effect is also called *oscillation* or *data looping*.

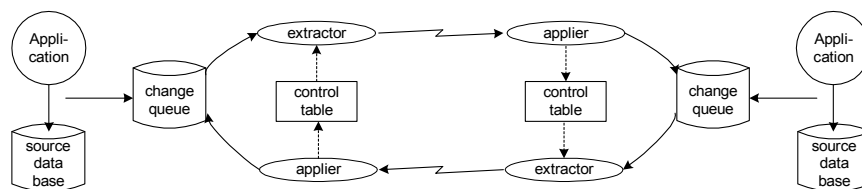
To understand why ping-ponging can occur, consider a replication engine that is being driven by an Audit Trail created by a transaction manager. Whenever a change is made to the database, the transaction manager will enter a description of that change into the Audit Trail. It doesn't care where the change came from. The data replication engine, unless told otherwise, will dutifully replicate that change to its target node, even if it just came from the target node.

Rule 14: *Block the ping-ponging of data changes in a bi-directional replication environment to prevent database corruption.*

In order to prevent ping-ponging, the replication engine must be able to determine the source of the change and replicate only those changes that were made by the local applications. Alternatively, all changes can be replicated back to the source node and ping-ponged

changes filtered out at the source node. There are several ways that this might be implemented²¹, including the following:

- **Partitioning:** If the database can be partitioned so that each data item is owned by one and only one node, and if only the owner can update its own data items, then the replication engine can be implemented to replicate only changes to those data items owned by its node.
- **Data Content:** In many cases, the row or record may indicate the source of the change. This might be, for instance, a source node id. In such a case, the data replication engine will not replicate changes originating at other nodes.
- **Control Table:** If there are no other means for the replication engine to determine the origination point of a change, then it can track remotely originated transactions via a Control Table that it maintains, as shown in Figure 3-7. As an Applier receives transactions and applies them to the target database, it enters a transaction id for that transaction into the Control Table. Likewise, as an Extractor at that node reads transactions from the Change Queue, it will check each transaction to see if that transaction's id is contained in the Control Table. If the Extractor finds the id, then it will not replicate that transaction.



Ping-Pong Avoidance
Figure 3-7

²¹ Strickler, G.; et al.; “*Bi-directional Database Replication Scheme for Controlling Ping-Ponging*,” United States Patent 6,122,630; Sept. 19, 2000.

Data Collisions

Because asynchronous replication modifies the target database some time after the source database has been modified, there is a time interval during that the data item that was modified is not accurately represented in the target database. This time is called the *stale time* and is predominantly the replication latency time.

Should an application modify a data item while that data item is stale, a *data collision* occurs. The source node is in the process of sending its latest change to the target node while the target node, acting as a source node, is replicating its latest change. Both changes are different and probably are both wrong. The database at this point has been corrupted and must be fixed.

Clearly, the frequency of data collisions will increase proportionally with the stale time. The stale time is predominantly the replication latency time of the replication engine as described in more detail in Chapter 9, “*Data Conflict Rates*.” Therefore,

Rule 15: *Minimize replication latency to minimize data collisions.*

There are two ways to solve the data collision problem. One way is to avoid it. The other is to detect data collisions and resolve them.

Collision Avoidance

There are configurations that are not susceptible to data collisions. Among them are:

Partitioned Database

As suggested above as a cure for ping-ponging, partitioning the database will eliminate collisions. To review, each partition in a partitioned database is owned by one and only one node. Only the owner of a partition may modify data in that partition. Since all application changes to a data item are made by only one node, there are no data collisions.

Partitioning may take many forms. It may be based on the value of a particular field. Perhaps only local sales offices can make changes to orders originating in their territory. A cell phone system may route messages to different nodes based on the first digit of the called or calling number. Partitioning might be based on customer and so on.

Partitioning might also be time-based. In a “pass the book” strategy, ownership of a database might pass from one node to another based on some criteria. An international brokerage firm might have the ownership of its database follow the sun, passing the database from office to office as the 24-hour day progresses. Alternatively, the ownership might be rotated between nodes on a schedule so that each node can process change requests that it has batched.

In any event, partitioning ensures that there is only one owner of a data item at any point in time. Therefore, there can be no data collisions.

Synchronous Replication

If partitioning is not feasible, and if data collisions must be avoided, then synchronous replication can be used. It is described in more detail in the next chapter. Synchronous replication ensures that all copies of a data item across the network are locked before any changes are made. Then all changes are simultaneously made before the data items are unlocked. In this way, changes at the target nodes are changed in synchronism with the changes at the source node; and no data collisions can occur.

Note that with synchronous replication, an application that is trying to modify a data item that is being changed by another application anywhere in the network must wait for the lock on that data item to be released before it can make its change. If asynchronous replication is being used, this application can make its change while the other application is making its change, and a data collision will result instead. As Jim Gray noted,²⁰

Rule 16: (Gray's Law) *Waits under synchronous replication become data collisions under asynchronous replication.*

Collision Detection

If data collisions cannot be avoided, then one must either decide that they are unimportant and will have no adverse effect or that they must be detected and resolved.

There are several existing techniques for detecting a collision. They are generally based on ensuring that the row or record to be changed on the target database is the same version that was changed on the source database:

Before-Image Comparison

If the replication engine sends before images as well as after images, then the Applier can compare the current state of the row or record on the target node to the before image of the record or row that was modified at the source. If they are the same, then a collision has not occurred; and the after image may be applied to the database.

Versioning

A single field indicating the original version of the source database record or row that is being modified can be sent with the change data. The Applier can then verify that the target record or row is the same version. There are many ways to check versions, such as a date-time stamp, an embedded version number field or column, or the CRC code associated with the row or column.

Collision Resolution

The resolution of data collisions is highly application-dependent, and the rules for resolution may be different for different fields. Each application must be carefully analyzed, and the business rules for resolving the results of a data collision for each field or column in each file or table must be determined. These rules may include the following.

Generic Algorithms

Many data collisions can be resolved by broad generic algorithms. For instance:

- always use the change from a designated master node.
- always use the change with the most recent or least recent time stamp or with the highest or lowest version number.

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

Data Field Specific Algorithms

One may be able to establish resolution algorithms that are specific to certain data types and/or data fields. For instance:

- for date/time fields, always use the earliest or latest date/time.
- for numeric fields, use the minimum, maximum, average, or some other function.
- for text fields, application-specific rules may be able to be formulated.
- for fields that have no real application significance, ignore the collision.
- for fields that may have no local significance, ignore the collision.

Relative Replication

For many numerical fields, such as counters, accumulators, and dollar quantities, relative replication may be used. The change in the numerical value can be computed by comparing the before and after images, and then that change can be applied to the field.

Care must be taken when applying relative replication. Only *commutative* operations may be replicated. Commutative operations are those that can be applied in any order and can still achieve the correct result. Addition and subtraction are commutative and may be replicated as differences, as described above. Multiplication and division are commutative and may be replicated as factors. For instance, if node A divides a data item by 5, and if node B multiplies it by 2, then node A can replicate a $\div 5$; and node B can replicate a $\times 2$.

However, addition/subtraction and multiplication/division are not commutative ($10+2\times 5$ is not equal to $10\times 5+2$). Furthermore, if a field is subject to both addition and multiplication, one cannot determine

from the before/after images whether the field was modified by addition or by multiplication. Therefore, the replication engine will not know whether to replicate differences or factors.

One problem with relative replication has to do with aborts. If commutative operations are applied relatively, then the operation can be saved as an effective before image. If the transaction is aborted, then the operation is simply reversed – i.e., an add becomes a subtract, a multiply becomes a divide, and so on. These are called *symmetric* operations. However, there are many operations that are *asymmetric* and cannot be easily backed out. The selection of the latest date or the maximum value of a numeric field are examples of asymmetric operations.

Fuzzy Replication

Fuzzy replication is the term given to a replication mode that re-performs a failed operation by mapping it to another operation in a defined way. For instance:

- If the operation is an insert that has found that the data record already exists, then it can be converted to an update using a mix of the above rules.
- If the operation is an update, and if the record does not exist, then it can be ignored (give precedence to the presumed delete with that it collided). It also can be converted into an insert, depending upon business rules.
- If the operation is a delete, and if the record does not exist, then it can be ignored. Alternatively, it may be converted into an insert with the relative difference applied.

Note that some of these sequences may repeat due to additional collisions, though with less and less likelihood as the sequence goes on. For instance, a collision that causes a data item to be deleted and then reinserted may continue if the deletion is retried in the face of another update.

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

Manual Resolution

If all else fails, the data collision must be sent to a human operator for manual resolution. In this case, the result will be very much dictated by business practices.

Failures and Recovery

Failover

One of the primary purposes of data replication is to provide continuing service to the users even in the presence of a node or network fault. What makes this possible is that following a failure, there is always at least one node still functioning in the network; and that node has access to a current copy of the database. The downed applications can move to surviving nodes. We call this procedure *failover*. Failover procedures vary somewhat according to the type of failure. Reference should be made to Figure 3-2.

Source Node Failure

- **Unidirectional Replication:** In the event of a source node failure, the backup node has a reasonable up-to-date copy of the database, albeit with the possibility of uncompleted transactions, that must be aborted. This is because the backup node may not have received certain transactions or their commits, that were still in the replication pipeline at the time of the source node failure. Normal processing may be resumed as soon as the users have been switched from the primary node to the backup node and as soon as the incomplete transactions have been aborted (to release locks that those transactions are holding). This switchover need take only a few seconds if the communication infrastructure is in place. Once switched over, users should each check that their last transaction has been applied to the database. Transactions lost in the pipeline should be resubmitted.

- **Bi-Directional Replication:** Whether bi-directional replication is being handled via partitioning or is active/active, only the users on the downed node are affected. All users on the surviving nodes are unaffected. As soon as the downed users are switched to a surviving node and check their last transactions, normal operations continue. If the increased load on the surviving nodes is a problem, load shedding by terminating non-critical applications may be in order.

Target Node Failure

- **Unidirectional Replication:** If the backup node fails, there is no impact on users. The users, who are all connected to the primary node, continue to be provided with full application services.
- **Bi-Directional Replication:** Users at the surviving nodes are unaffected. Failover for the users at the failed node is identical to that described above for a source node failure.

Network Failure

Should the communication channel fail between a pair of nodes, the nodes are left operational; but replication will cease. During this time, the nodes will queue their database changes to their Change Logs for use during later restoration.

- **Unidirectional Replication:** In a unidirectional replication environment, there is no further action to take. Users remain active on the primary node.
- **Bi-Directional Replication:** There are many strategies that may apply to failover following a network failure in a bi-directional environment. One strategy is to allow users at isolated nodes to continue their normal functions. The replicating engines will queue all database changes to the isolated node for later restoration.

- However, during the network down time, if the network is not partitioned, there is the possibility that many data collisions will occur and will need to be detected and resolved at restoration time. If collisions are a problem, then isolated nodes should be considered to be failed nodes; and users should be switched to other nodes as described above.

Restoration

Restoration of full services following the return to service of a failed or isolated component is accomplished by resynchronizing all database copies in the network. During the failure period, all active nodes were queuing their database changes for later replication. Now, they will each begin replaying these changes to the previously downed or isolated nodes (that can include two-way replication if two or more nodes were isolated by a network failure). When all change queues have been drained to an acceptable level from a replication latency viewpoint, and when the users have been switched back to their home nodes, normal system operations resume.

In order to replay changed data, the source replication engine's Extractor must know where successful replication left off. It does this by keeping track of the Change Log's oldest transaction still outstanding at the time of the failure (to support this, an Applier might return a confirmation to the Extractor when it has successfully applied a transaction). Periodically, the Extractor may checkpoint this restart point to disk.

When a replay is required, the replication engine will begin sending transactions starting at the replay point. Normal operations can continue at that node since subsequent database changes will simply be added to the tail of the Change Queue. When the Change Queue has been drained to the point that replication latency has returned to a normal level, synchronization of the database copy has been completed.

There are, however, some critical issues associated with replay:

Duplicate Transactions

It is quite possible that transactions that have already been applied to the target database will be replayed again during restoration. There are two reasons for this. One is due to the fact that transactions are interleaved. Starting replay at the oldest open transaction implies that there are more recent transactions that have completed and that will be replayed. Also, if a disk checkpointed restart point is used, there may have been significant transaction activity following the checkpointed replay point; and that activity will all be re-replicated.

In most cases, transactions cannot be duplicated. Most are not *idempotent* – that is, multiple applications of the same transaction will yield different results. To correct this situation, duplicate transactions must be detected and ignored. One way to do this is for each Applier to maintain a persistent record at the target node of all transaction ids that have been successfully applied to the target database. By including updates to this table within the scope of the target transaction, a transaction id will be entered only if the transaction successfully committed.

On replay, the Applier will check each incoming transaction against this table to see if the transaction is a duplicate and will discard it if it is.

Data Collisions

If replication is unidirectional or if it is bi-directional partitioned, then there is no problem with data collisions while the downed database is being resynchronized. However, if replication is active/active, and if a network failure isolated one or more nodes that remained active during the failure (i.e., users were not switched over to connected nodes), then data collisions may occur during the network down time. In effect, the stale time has been extended from the normal replication latency time to the network failure time.

These data collisions are no different than normal data collisions that occur during normal asynchronous replication stale time, except