Highly Available Cluster: A Case Study

Alain Azagury, Danny Dolev, Gera Goft,
John Marberg, Julian Satran

IBM Israel Science and Technology
Matam — Advanced Technology Center
Haifa 31905, Israel
E-mail: azagury@vnet.ibm.com

Abstract

The methodology and design of a system that provides highly available data in a cluster is presented. A *Highly Available Cluster* consists of multiple machines interconnected by a common bus. Data is replicated at a primary and one or more backup machines. Data is accessed at the primary, using a location independent mechanism that ensures data integrity. If the primary copy of the data fails, access is recovered by switching to a backup copy. Switchover is transparent to the application, hence called *seamless* switchover. The fault model is fail-stop. The entire cluster is resilient to at least single failures. Designating data as highly available is selective in scope, and the overhead of replication and recovery is incurred only by applications that access highly available data. An experimental prototype was implemented using IBM† AS/400† machines and a high-speed bus with fiber-optic links.

1 Introduction

The system presented in this paper provides highly available data through replication in a cluster of machines. The project has three main goals: (a) to demonstrate the feasibility of highly available data on interconnected machines with minimal hardware requirements; (b) to develop cost-effective methodologies and mechanisms for high availability of data; and (c) to implement a prototype as a vehicle for performance measurements.

A *Highly Available (HA) Cluster* consists of two or more machines interconnected by a high speed bus. Each machine has its own private disks. Workstations are connected to the cluster through a local area network (LAN). The failure of a machine causes both it and its disks to become unavailable. Thus, to achieve high availability we use data and process redundancy. The fault model for both data and processes is the *fail-stop* model. The design ensures that the entire cluster is resilient to (at least) *single failures*.

The HA Cluster introduces a new concept: *resilient groups*. A resilient group is a set of data collections (libraries) that are replicated at a primary and at one or more backup machines. Access to a group is at the primary, via a location independent mechanism that uses global name resolution and handles data replication and integrity. When the primary copy fails, access is recovered by switching to a backup copy of the group. switchover does not involve any action by applications that use the resilient group, i.e. it is transparent to the application, and hence called *seamless switchover*. The seamless switchover mechanism introduces a new way of using database journals to disseminate data, synchronized with the control state of the primary copy.

The system defines a clear interface between the application and the data services it uses. The location of data services is transparent to the application, and the services are accessible from any machine in the cluster. Therefore, by replicating the code of the application and its working environment on several machines, the application becomes highly available. When the machine where the application runs fails, the user can restart it on any other machine in the cluster. Data replication ensures that no committed transactions are lost and therefore only the last non-committed transaction needs to be re-entered. Applications that are suitable for this approach are generally classified as on-line transaction processing. We assume that such applications are invoked from a workstation connected through a LAN to one of the machines in the cluster.

Our design provides high availability with selective scope. The user (or administrator) explicitly designates which libraries are highly available, by including them in a resilient group. The performance overhead of replication and recovery mechanisms is incurred only by applications that access resilient groups. Other applications are not affected.

We assume that applications invoke system services via existing standard interfaces. High availability mechanisms are designed to be transparent to applications that use such interfaces. Thus, high availability is an attribute of the environment, not of the application. Moreover, the application developer need not be aware of high availability when designing or implementing the application. Existing applications work with highly available data without modification.

The intention from the outset has been to build a small cluster (at most tens of machines) with a specific communication architecture. Scaling up the design for a larger number of machines or for a more general distributed environment was not contemplated. Our communication architecture provides an environment that allows for much simpler cluster management protocols than those in previous systems such as [1, 4, 14, 18, 19]. By tailoring the algorithms for the particular environment we gain increased performance efficiency.

Our system is related to other distributed and highly available systems [2, 3, 9, 11, 15-17, 20, 21]. In [5], many more such systems are surveyed. Previous projects either assumed shared disks, or covered only part of the span of our project. Focus was either on file replication or on transaction processing, without offering a seamless solution. Our solution is transparent to applications, while offering seamless switchover.

† IBM and AS/400 are trademarks of International Business Machines Corporation.

0363-8928/94 $3.00 © 1994 IEEE

We have implemented the mechanisms described in this paper in an experimental prototype cluster of IBM AS/400 machines [13]. The hardware configuration consists of two AS/400 model D60 machines connected by a high speed bus with fiber-optic links. Performance measurements have been conducted, using a variant of the TPC-B benchmark[10].

The organization of the paper is as follows. Section 2 gives an overview of cluster organization. Section 3 describes mechanisms for resilient data access, and Section 4 mechanisms for failure recovery. Section 5 discusses cluster management. Section 6 presents performance measurements of the prototype. A Summary is given in Section 7.

2 Cluster Organization

This section gives an overview of the organization of the cluster, from the physical and functional perspectives.

2.1 Physical Organization

The physical organization of a typical HA cluster is shown in Figure 1. The cluster consists of several machines interconnected through a common bus.

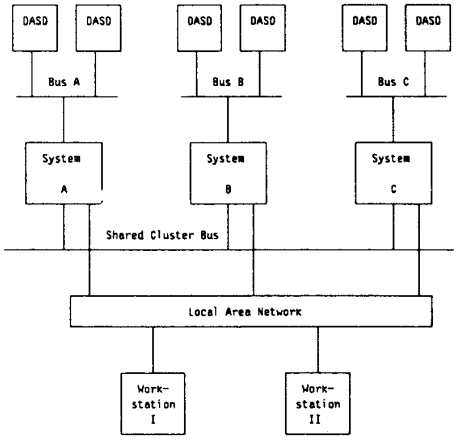

Figure 1. Physical organization of the highly available cluster

The only interconnection available is among the processors. Sharing or switching disks among machines is not supported. Therefore, multiple copies of the data are maintained on different machines to make the cluster resilient to faults.

All communications among machines in the cluster are performed over the cluster bus. A local area network (LAN) is used only to connect terminals and workstations to the cluster. The LAN could be potentially exploited to support application resilience, using a mechanism to switch terminal sessions transparently among machines.

The cluster bus is a dedicated AS/400 system I/O bus [13] with fiber-optic links. Actually, a pair of buses is being used, with a separate power domain and bus-master for each bus. The bus attachment hardware provides link redundancy for each bus and between the pair of buses. It is therefore assumed, in the context of the single failure model, that no faults are incurred by the cluster bus medium itself.

The bus architecture guarantees that there will be no partitions in the cluster. All functioning machines can communicate directly with each other at all times. The communication graph is thus a complete graph. In the context of our work and in light of the specific architecture, it is assumed that the bus mechanisms guarantee timely delivery of all messages.

These assumptions have implications that are critical to synchronization and recovery, as discussed in Sections 4 and 5.

2.2 Functional Organization

The cluster comprises several objects and functional components. An example configuration with three machines is shown in Figure 2.

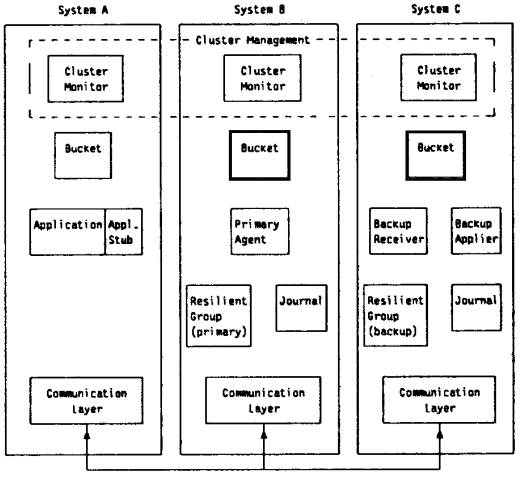

Figure 2. Functional organization of the highly available cluster

A *Resilient Group* is a set of data collections (libraries) that are considered a unit for purposes of high availability, and for which data resilience against single failures is guaranteed. For example, a resilient group could comprise the libraries of some database. The group is replicated on a number of cluster machines. One of the replicas is used for data access, and is called the *primary*. The other replicas are called *backups*. A resilient group can be accessed from every machine in the cluster. The example in Figure 2 shows a resilient group with replicas on machines B and C. In the context of a specific group, the machines where the primary and backup replicas reside are called primary and backup machines, respectively.

A *Journal* is a log of database operations on objects of a resilient group. All the objects of a given group are journaled in the same journal, and a separate journal is maintained for each group. A copy of the journal exists on each machine containing a replica of the group. Operations are journaled at the primary, and the journal entries are shipped to the backups. Serialization of the journal preserves the order of operations at the backups.

A *Bucket* is an object that constitutes the definition of a resilient group. The bucket is used by data access mechanisms and by cluster management. It contains the names of the primary and backup machines, the names of the libraries and journal, and a log of the activities in the group (e.g., which applications work with which objects). The bucket exists on every machine of the cluster, even if there is no replica of the group on the machine.

An *Application Stub* is a system module providing cluster interface for database operations. An operation issued against a resilient database is intercepted by the stub, who ships the request to the primary site of the database. The function of the stub is transparent to the application.

A *Primary Agent* is a process on the machine containing the primary replica of a database, whose role is to execute database operations shipped from application stubs. The agent also forwards the operations' journal entries to the backups of the database. There is a separate agent for every application process.

The *Backup Receiver* is a process on a backup machine, that receives journal entries from the primary and deposits them in a local copy of the journal. There is a separate backup receiver for every backup journal.

A *Backup Applier* is a process on a backup machine, that applies the journal entries onto the local replica of the database. There is a separate backup applier for every backup journal.

A set of *Cluster Monitor* processes, one on each machine in the cluster, jointly perform distributed cluster management. The monitors maintain consistency and availability in the cluster, keep track of all cluster entities, and react to events, such as machine failure or configuration change request.

The *Communication Layer* provides a set of services enabling cluster entities on different machines to exchange messages, ship operations, and broadcast over the shared cluster bus. All communications among machines in the cluster are carried out over the shared cluster bus.

The buckets, application stubs and agents provide a location transparent name service, similar to previous systems such as [8, 11].

3 Replication and Remote Data Access

This section discusses the characteristics of the resilient database system, and the mechanisms for data replication and remote access.

3.1 Database Characteristics

The database has the following characteristics.

- Access to the database is through a well-defined set of interfaces.
- The database management system allows a hybrid transactional/non-transactional mode of operation. Some applications may access the database in a mode that guarantees atomicity, consistency, isolation and durability (the ACID property), while other applications may read dirty data (non-committed modifications) and perform single updates.
- The transactional mode of operation uses locking. While executing a transaction, an application will lock any items it attempts to read or write (there are several levels of locking). Thus, an attempt to commit a transaction will not fail because of conflict with another application. It is assumed, generally, that applications will not provide a recovery action from a commit failure, since this is considered an extremely rare case.
- Most database operations return some non-trivial feedback information, e.g., for record update, the record's length, ordinal number, format name, key information, etc.
- Modifications to the database are logged in a *journal*. The journal has a major role in providing the transactional behavior of the database. Access to the journal is serialized by means of an exclusive lock, and journal entries are guaranteed to reach stable storage before the corresponding database modifications do. The journal may also be used for recovery purposes. An old copy of a database may be brought up to date by *applying* the modifications logged in the journal. Journal entries are idempotent, i.e., re-application of the same entry does not further modify the database.
- Journal entries are assigned a monotonic ascending sequence number.

3.2 Accessing Resilient Data

The flow of control for a typical operation that modifies a resilient database (e.g., an update of an existing record, or a write of a new record) is illustrated in Figure 3.

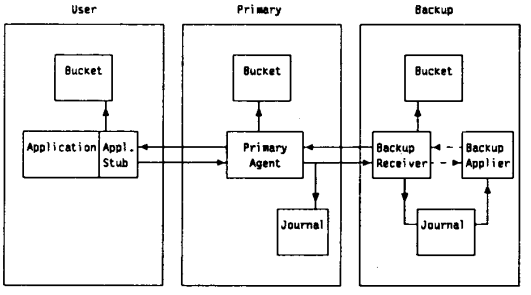

Figure 3. Flow of control for a database modification operation

The following protocol is performed.

1. The stub intercepts the execution of the operation at the application machine.
2. The stub ships the operation request to an agent at the (current) primary machine of the database.
3. The primary agent executes the operation. Any journal entries inserted into the journal are broadcasted to all the backup receivers.
4. The backup receivers acknowledge receipt of the journal entries. Notice that journal entries arrive at the backups in the correct order, as the communication layer guarantees that messages are delivered in the order they were sent.
5. The primary agent returns the operation feedback to the application stub.

6. The stub returns control to the application.

The backup applier asynchronously applies the journal entries of the backup journal onto the backup replica.

When multiple backups exist, the agent may return to the application stub (step 5) as soon as the first acknowledgement is received, without affecting the resilience to single failures.

Operations that do not modify the data are not journaled and do not affect the backups. However, modifications to the cursor and locks (if any) are returned to the application as feedback, to be used during recovery from data failure.

3.3 Exploiting Backup Replicas

The backup replicas of a database can be exploited for various administrative tasks, in order to reduce the load on the primary system. For example, taking a snapshot of the database (e.g., for backup to a tape). This can be done without compromising the ability of the backup system to become primary in case the current primary fails. Since the backup replica is not on the data access path, taking a snapshot does not add overhead to data access. If the backup replica is not modified while the snapshot is being taken, a simple snapshot protocol can be used, as follows.

1. The administrator issues a request for a snapshot.
2. The snapshot request is routed to the primary system.
3. When the data reaches a consistent state, the primary ships the snapshot request to one of the backups.
4. The backup receiver marks the snapshot point in the journal.
5. The backup applier applies all the journal entries preceding the snapshot point. It then freezes application of further changes.
6. When all changes to the data are applied, the snapshot is taken.
7. When the snapshot is complete, the backup applier resumes normal operations.

Notice that receiving changes at the backup and applying these changes are two separate activities, performed by independent threads of execution. Hence, taking a snapshot has no impact on the flow of changes towards the backup. If the primary replica fails while the backup is being used for a snapshot, the backup can still become primary, since all the changes made at the (old) primary are journaled at the backup. Recovery may take longer, but data resilience is guaranteed.

4 Data Failure and Recovery

This section discusses the problems encountered during data failure, and the mechanisms used for recovery.

4.1 Failure Scenario Example

Consider the scenario illustrated in Figure 4. We use this example to emphasize the integrity problems caused by data failure. Four applications are concurrently accessing a database whose primary copy resides on machine *A*; the backup is on machine *B*. Notice that the applications may run on any machine in the cluster, including the primary or backup machine of the database.

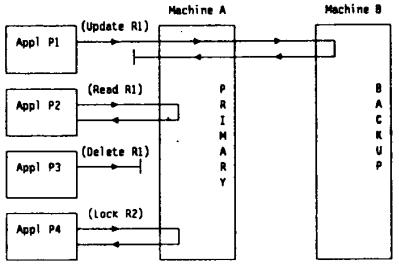

Figure 4. Cluster state during failure

1. Application *P1* issues an *update* request for record *R1*. The primary *A* updates the record and successfully sends a journal entry reflecting the change to the backup machine *B*. A failure occurs on the primary before it returns the feedback of the operation to the application.
2. Application *P2* issues a *read* request for record *R1*. The primary returns to the application a record that reflects the last update made by application *P1*. The record is received by *P1* before machine *A* fails.
3. Application *P3* issues a *delete* request for record *R1*. The primary *A* fails before receiving the request.
4. Application *P4* issues a *lock-for-update* request for record *R2*. The operation is successful and the feedback arrives at the application before the primary *A* fails.

It is assumed that all applications survive the failure, i.e., either none of the applications run on machine *A*, or *A* continues running despite damage to some part of the primary copy of the database. The backup *B* becomes the new primary. The following occurs immediately after the switchover.

1. Application *P1* re-issues the *update R1* request to the new primary *B*. The recovery protocol must detect that the original request was successful, as its effects are already known elsewhere (application *P2* has already read the modified record). The operation feedback that would have been sent by the old primary needs to be recreated and returned to the application. This can be done using the journal entries that were sent to the backup *B* prior to the failure.
2. Application *P2* continues, and now issues a new *lock-for-update* request for record *R2*. The record was locked by application *P4* when *A* was still primary. The protocol needs to prevent granting the lock to *P2*, otherwise application *P4* will fail while trying to update the record, i.e., the switchover will not be seamless.
3. Application *P3* re-issues the *delete* request. The protocol must detect that the original request was not completed, and the operation must be re-executed.
4. Application *P4* still assumes that it holds the lock on *R2*. The protocol must ensure that this lock is acquired on behalf of *P4* on the new primary *B*.

Locks are guaranteed to survive a failure by logging them in the feedback returned to the application stub from each

operation. During switchover, application stubs send the feedback information to the new primary, who reacquires the locks before activity on the database is allowed to resume.

To distinguish between the cases of applications *P1* and *P3*, the sequence numbers of the journal entries of the operation are returned to the application stub as part of the feedback. Upon switchover, the application stub re-issues the operation to the new primary together with the last sequence number it has available. At the new primary (previously the backup), the sequence number is compared with that of the most recent entry of the same application in the local journal. A mismatch indicates that the operation was completed successfully on the old primary, but the feedback was lost.

4.2 Recovery Coordination

Upon detecting a failure of the primary copy of a group, the cluster monitors coordinate recovery with the processes involved, thereby effecting a *seamless switchover*. The primary copy of the group is invalidated, and the first backup is made primary. The recovery protocol consists of the following steps.

1. The monitor on each machine locks the group in the local bucket, to suspend any access to the group from applications.
2. The monitor of the failed primary locks the journal, to suspend sending new journal entries to the backups. It then notifies the monitor of the first backup that the journal is locked. (If the primary machine has crashed, this step is skipped).
3. Each monitor updates the local bucket with information of the new primary.
4. When the monitor of the first backup learns that the journal is locked, it instructs the backup receiver to apply all journal entries and force the journal to stable storage.
5. Each monitor instructs the local stubs to send to the new primary the feedback of previous operations. This will be used in recreating the database state at the new primary.
6. An agent is created at the new primary for every active application. The new agents work with the local backup receiver to recreate a state equal to that which existed at the failed primary. The backup receiver then terminates.
7. The monitor of the new primary notifies all other monitors to release the group lock. It also notifies the monitor on the failed primary to release the journal lock. Applications may now resume operations on the group. The stubs now route the operations to the new primary.

Notice that application stubs or agents that were asleep during the switchover may attempt to use the old primary and backups. This can be detected by consistency checks. The operation is aborted and retried using the new setup.

4.3 Reinstating State Information

For each application using the database, the cluster maintains *state information* at the primary machine (position, locks held, commit information, record modifications, etc.). Some attributes of the state information are kept by the database management system (record locks, record contents, etc.), and others are kept by the primary agent (e.g. position).

Upon failure of the primary, one of the backup replicas is designated by cluster management as the new primary for the database. The state of the database needs to be recreated at the new primary, so that surviving applications can resume activity. The goal is to make the switchover completely transparent to the applications, except, perhaps, for a short delay.

For every database operation that was in progress at the time a failure occurred, it must be determined whether the operation needs to be re-executed, or whether it has actually completed executing on the old primary. In the latter case, the feedback from the operation needs to be recreated at the new primary and returned to the application stub.

We provide mechanisms to efficiently replicate state information with minimal impact on normal transaction throughput, and to determine the status of operations that were in progress at the time of failure. Also devised is a mechanism to regenerate in a simple manner the operation feedback that was about to be returned to application stubs when the failure occurred. The latter is particularly important for operations whose feedback cannot be determined from the journal entry alone. We call such operations *critical operations.*

The state information comprises two parts: (a) application related and (b) database related. Application related state information is relevant only for recovery of the specific application (e.g. cursor position), whereas database related state information is relevant also for recovery of other applications (e.g. updated values of records).

Application related state information is returned to the application with the operation feedback. Database related state information is broadcasted to the backup machines while the operation is being journaled.

For a journaled operation, the journal sequence number of the entries generated by the operation is returned to the application. The last sequence number received by an application will help determine whether or not the operation that was in progress at the time of failure has completed successfully.

Incorporating these techniques, a typical modification operation on the database proceeds as follows:

1. The database request issued by an application is intercepted by the application stub.
2. The stub ships the operation to the (current) primary machine for the database.
3. The primary agent executes the operation. Any journal entries generated by the operation are broadcasted to the backup receivers. The sequence number of the journal entries are recorded. The journal is locked exclusively while the new entries are inserted and broadcasted to the backup receivers. The lock is released while waiting for acknowledgements, to allow overlapping of broadcast operations.
4. The backup receivers acknowledge receipt of the journal entries.

5. The agent returns the operation feedback to the application stub, together with state information: cursor position; record locks; and journal sequence numbers.
6. The application stub saves the state information and returns control to the application.

The backup applier asynchronously applies the journal entries onto the backup replica. When encountering a journal entry of a critical operation, the backup applier blocks until a new journal entry is received from the same application. Blocking on the journal entry enables re-executing the critical operation on another machine if the primary fails, thereby effectively and easily regenerating the operation feedback. The arrival of a new journal entry indicates that the application safely completed the critical operation.

When the primary fails, the following is performed.

1. Cluster management designates one of the backup machines as new primary for the group.
2. Application stubs are instructed to send state information to their (new) agent at the new primary. This information is used, in conjunction with the journal entries, to restore the database state and the agent state.
3. For each database operation that was in progress at the time of failure, the journal entry sequence number received from the application stub is compared with the last sequence number received from the old primary agent. This determines whether or not the pending operation was completed at the old primary.
4. For each completed operation, the operation feedback is returned to the application stub. For a non-critical operation, the feedback is recreated by the backup receiver from the journal entry. For a critical operation, the operation is re-executed by the new agent, thereby effectively generating the feedback as well as applying the changes onto the local replica.

The recovery mechanism does not abort uncommitted transactions. Each transaction is resumed at the point where it was interrupted by the failure. Applications resume activity as soon as the new primary agents are created and the backup applier (or the new primary agent, in case of critical operations) has applied all the journal entries that were received from the primary.

If there are several backups, the primary agent sends the journal entries to all backup receivers, but need wait only for the first acknowledgement before proceeding with further operations. In this case, the journals of the different backups must be reconciled before recovery is completed.

To provide acceptable response time and prevent a single sluggish application from blocking all other applications, one needs to compromise transparency in favor of efficiency. Blocking operations must be timed-out, and recovery needs to be provided for time-out situations.

For critical operations, if the application that issued the operation does not execute a new operation within a predefined time frame, the backup applier should resume execution. Otherwise, the backup applier could fall behind the primary, rendering potential recovery from failure too slow. An obvious risk, in case of a primary failure, is that the application which issued the critical operation may have to be aborted if it did not receive the feedback.

During the recovery phase, the new primary agents need to coordinate with the backup applier in order to bring the database up to date. This must be done before activity on the database is allowed to continue. Again, an agent failing to respond to the backup applier within a reasonable time frame will be ignored by the backup applier. This could potentially result in loss of feedback information for a pending operation, forcing the agent to return an error to the application.

4.4 Journal Recovery

A machine that rejoins the cluster after it has failed must recover its local database replicas. If the failure occurred while the machine was the primary site of a database, recovery entails deleting an undetermined number of journal entries which, due to the failure, were not replicated at any backup site. Normally, such entries are identified by simple comparison with the journal of the current primary replica.

A mechanism is provided that bounds (a priori) the number of entries that need to be deleted. This is useful when the latest primary replica of the database is damaged (due to subsequent failure) and cannot be consulted. In this case the recovered journal has to be backed off to an old, consistent state.

A counter, called the *uncertainty counter*, is added to the journal. The counter is incremented by a primary agent before it broadcasts a block of journal entries to the backup receivers, and is incremented when all acknowledgements are received. If, as a result of the increment, the counter reaches a specified threshold value, the primary agent will not release the journal exclusive lock until it has received acknowledgments from the backup receivers. This will prevent new operations that attempt to modify the database from progressing. A typical modification operation now proceeds as follows:

1. The operation is intercepted by the application stub.
2. The stub ships the operation request to the (current) primary.
3. The primary agent executes the operation. Before modifying the contents of the database:
   - The journal is locked exclusively.
   - The uncertainty counter is incremented.
   - The block of journal entries is inserted in the journal and broadcasted to the backup receivers.
   - If the value of the uncertainty counter is less than the threshold, the exclusive lock is released.

   (Notice that database modifications are allowed to reach stable storage only after the journal entries do).
4. Backup receivers acknowledge receipt of the journal entries.
5. The primary agent decrements the uncertainty counter. If the journal lock is still held, it is released at this point.
6. The primary agent returns the operation feedback to the application stub.

7. The application stub returns control to the application.

Let n denote the threshold value of the uncertainty counter. When an "old" primary replica is recovered without having access to the current primary replica, at most n journal entries need to be deleted. The threshold value should be tuned such that it will seldom be reached, thereby maximizing the throughput in the journal.

5 Cluster Management

Cluster management is performed distributively by a set of cluster monitors, one on each machine.

Cluster management is event driven. Events may be caused by failures or initiated explicitly by the system administrator. Events are created by arrival of messages at the monitors. Such messages come from other monitors, agents, application stubs, backup receivers, and the communication layer.

Following are some of the events that require action by the monitor.

- Machine failure.
- Group failure (including library, database or journal failure).
- Application failure.
- Communication failure.
- User or system administrator initiated events, such as: joining a machine to the cluster, detaching a machine from the cluster, and creation or deletion of group objects.

This section discusses some of the mechanisms used in cluster management: clock synchronization, synchronous update, and heartbeat.

5.1 Clock Synchronization

The machines in the HA cluster have synchronized clocks. This is necessary in order to schedule events at the same time on every machine in the cluster. For example, changing the cluster configuration tables requires global synchronization, since these tables have to be identical on all machines at all times.

It is assumed that each machine has access to a physical hardware clock. Each machine also maintains an adjustment register. The *logical clock* time is the sum of the values of the physical clock (over which there is no control) and the adjustment register.

The clock synchronization algorithm is essentially that of [12]. It is outlined here for the sake of completeness. The algorithm is based on the following simple observation. If there are no faulty machines, one machine can act as a synchronizer and periodically broadcast a message with its current time (the frequency of synchronization depends on clock drift). Each machine then adjusts its own clock, making a minor allowance, if necessary, for the transmission time of the message.

The protocol described below guarantees that the logical clock times of all machines will be kept sufficiently synchronized, and remain within a linear envelope of real time, even in the presence of any single failure.

Each machine uses a local timer which is set to a predetermined time for the next clock synchronization.

1. Upon expiration of the timer for the next clock synchronization, broadcast a message with the local logical time to all machines, and set the timer to the next synchronization time.
2. Upon arrival of a clock synchronization message, if the time received in the message is not equal to the next synchronization time ignore the message. Otherwise, forward the message to all machines; adjust the logical clock according to the received time; set the timer for the next clock synchronization; also, adjust any other timers that are based on the logical clock (e.g. the timer for the next synchronous update — see Section 5.2).
3. Upon arrival of a forwarded clock synchronization message perform the same as in step 2, without forwarding the massage.

The algorithm requires $O(n^2)$ messages for each clock synchronization, where n is the number of machines in the cluster, since each machine sends a message to all other machines. Notice that the probability that the timers of two machines will expire at the exact same moment is very small.

The time interval between synchronization points is a preset constant that depends mainly on the relative drift of the physical clocks and the message diffusion time.

Assuming single failures, forwarding the synchronization message once (step 2) guarantees that each system will receive at least one message in each round. Ignoring messages whose time stamp is too old protects against late arrival of messages.

5.2 Synchronous Update

Some events require cooperation among all monitors, and should not be handled by a single monitor. Examples are:

- *Machine failure.* There is no predetermined monitor in charge of detecting or handling the failure of another machine.
- *Group Creation.* The new group must be made available on all machines at the same time. Also, simultaneous operations on the same group definition must be globally serialized.

Such events typically result in updating the cluster configuration, which needs to be done consistently and simultaneously on all machines. The *synchronous update* mechanism [7] enables cluster-wide serialization of such updates. It guarantees that updates are performed in the same order and within a specified time window on every machine in the cluster. This allows machines to maintain a consistent view of the cluster. The mechanism relies on the accuracy of the logical clocks maintained by clock synchronization.

For efficiency reasons, not all events are handled by the synchronous update mechanism. The risk in not using this mechanism is that, at certain points, some monitors and some machines will already be updated, whereas others

may not yet be, thus introducing inconsistency into the system.

The design of our system carefully combines cluster-wide cooperation with single machine (primary machine) responsibility. Many of the events related to the operation of a primary machine can be disseminated asynchronously without any global serialization. For example: adding a new backup replica of a group (handled by the primary); group backup failure (handled by the primary); and group primary failure (handled by the first backup). The inconsistency in not receiving the event on time will not surface if the system design ensures coordination with the primary before taking certain actions. Hence, a machine or a process that did not receive a disseminated event will communicate with the source of that event, and thereby get updated.

Following is a description of the synchronous update mechanism. Each monitor maintains a list of pending update events. Each event is a pair (t, upd), where t is the time at which the update *upd* should be performed.

A timer associated with the event list is set to expire at the time of the next update. Whenever the logical clock changes as a result of clock synchronization, the timer is adjusted accordingly. When the timer expires, the monitor performs all the updates whose time is less or equal to the current time, and removes them the list. The timer is then set for the next update.

To schedule a new synchronous update, a monitor creates an event pair (t, upd). Let d denote the diffusion time of a message in the cluster. The time t is adjusted so that the event is scheduled at least $2d$ time units after the current logical clock time, and at least $2d$ time units before or after the next clock synchronization. This gives sufficient time to disseminate messages containing the event to all other monitors, and prevents scheduling events within a clock synchronization period. In such a period, clocks change their value, so some events might be scheduled at different clock times on different machines. A more complicated clock synchronization mechanism can eliminate this conflict, but it would introduce unnecessary overhead into the system.

Adding the new event to the list requires coordination among all the monitors. Similar to clock synchronization, the protocol is protected against any single failure by forwarding each message once.

1. The initiating monitor broadcasts the update pair to all other monitors and insert it into the local list. If the current time is too close to the next clock synchronization, the broadcast is deferred until after clock synchronization.
2. When receiving a message containing an update pair, the monitor verifies that the pair is not already in the local list, and that t is a valid future time (the latter protects against late arrival of messages). The monitor then inserts the pair into the local list and forwards the update message to all other monitors.
3. Upon receiving a forwarded update message, the monitor performs the same as in step 2, without forwarding the message.

The synchronous update mechanism requires $O(n^2)$ messages per event, where n is the number of machines in the cluster. By comparison, in our cluster architecture, asynchronous event handling requires only $O(n)$ messages, to notify the machines in the cluster, since each machine can always communicate directly with all other machines.

5.3 Heartbeat

During normal operations of the cluster, every monitor needs to be able to communicate with all other monitors. A *Heartbeat* mechanism [2] detects any disconnection among the monitors in the cluster. Monitors broadcast a special heartbeat signal at predetermined intervals (rounds), and acknowledge every arriving heartbeat signal. Not receiving a timely acknowledgement form some monitor indicates that there is a disconnection, entailing recovery action in the cluster.

If there are no assumptions on connectivity among the machines, a distributed heartbeat mechanism (i.e. no designated central authority) may require up to $O(n^2)$ messages per round, where n is the number of machines in the cluster. This is since every machine may have to send signals to all other machines in each round. This adds significant overhead to the communication layer, and message delivery within a short period of time cannot be guaranteed.

The $O(n^2)$ message bound can be improved to $O(n)$ if the communication medium connecting the machines has the following property, called *connection transitivity and symmetry (CTS)*: for any three machines i, j, and k,

- if i is connected to j, and j is connected to k, then i is connected to k.
- Connections are bidirectional.

Assume there is a disconnection between machines j and k. By the CTS property, every machine in the cluster is disconnected from either j or k. In other words, in the heartbeat algorithm, every machine will detect a disconnection.

Thus, it suffices that in each round only one machine will issue the heartbeat signals, requiring only $O(n)$ messages. In a realistic distributed environment, an $O(n)$ bound can be achieved by setting the time of the next heartbeat at each machine randomly within a certain window around the exact time. This minimizes the probability of two machines simultaneously initiating heartbeat signals.

The AS/400 System I/O bus has the CTS property, thus enabling us to design an optimal heartbeat mechanism.

A different heartbeat protocol, also requiring only $O(n)$ messages, can be based on the approach in [6]. There, processes are arranged in a virtual ring, and messages are exchanged among neighbors on the ring.

6 Prototype Performance

This section presents the results of performance tests conducted on the Highly Available Cluster prototype. Measurements focus on throughput overhead during normal application flow, and on switchover time in various cases.

The cluster used in the tests consists of two AS/400 model D60 machines, each with 64MB main memory. One machine has 3166MB of DASD storage, and the other has 1583MB. The machines are attached to a dedicated bus

via fiber-optic links. The cluster monitor processes run in a separate subsystem with 2MB dedicated storage.

The benchmark application is a variant of TPC-B[10]. Instead of a single set of four database files, there are four identical sets (16 files total). Each set can reside on a different machine. The benchmark application opens all 16 files, and for each transaction chooses randomly the set to be used. It is possible to obtain configurations where the primary and backup machines are defined independently for each set. Two different configurations were used in the tests:

1. A single group comprising all four sets. Application jobs initially have agents only on the machine defined as primary for the group, requiring agents to be created on the backup during switchover.
2. Two groups of two sets each, with a different primary machine for each group. Every application job has an agent on both machines, thus avoiding the overhead of agent creation during switchover.

Notice that the cluster consists of two machines only, so application jobs run either on the primary or on the backup machine of the data.

Two benchmark drivers were used: a batch driver and an interactive driver. The difference between the two is that the interactive driver sleeps for 10 to 20 seconds between TPC-B transactions to simulate "key-think" time.

6.1 Throughput Tests

Throughput tests were conducted using the batch driver. Although all tests consist of two time intervals, one for "burn-in" (to stabilize the system) and the other for measurements, there was a significant variance between successive runs, even when long (25 minutes) burn-in time was used. The seed of the random transaction selector was set to a constant, so that tests would become more uniform and stabilized.

For the results presented here, 5 minutes burn-in time and 5 minutes measurement time were used. Several successive runs were made for each test case, and the results of the last run (which were generally the best) were chosen.

Table 1 shows the transaction rate for different configurations. It also shows the degradation with respect to the non-cluster (single machine) environment.

Table 1. Throughput for the TPC-B variant

Test description	Transactions/second	Degradation
Single job, non-cluster environment	8.2	100%
Single job running on the primary machine, no backup. Overhead consists of database access via local communication with an agent.	6.25	76%
Single job running on the primary machine, with backup. Overhead consists of database access via local communication with an agent and sending of journal entries to a remote machine.	6.07	74%
Single job running on the backup machine. Overhead consists of database access via remote communication with an agent and sending of journal entries to a remote machine.	5.4	66%

With a single application running, and the primary being on the same machine as the application, the total CPU utilization of the primary machine is 60%. By comparison, CPU utilization is 45% when running the same application in a non-cluster (single machine) environment.

Table 2 shows the CPU utilization of various functions of the cluster.

Table 2. CPU utilization of individual cluster functions. Data taken from the *WRKACTJOB* display.

Function	CPU %
Primary Agent	17%
Backup Receiver	3%
Backup Applier	8%
Application	22%–36%†
Monitor	0.2%‡

† Depends on the number of applications (1–3).

‡ During normal operation (not during switchover).

6.2 Switchover Tests

Switchover tests were conducted using the interactive driver. Similar results were achieved when running one batch application vs. one interactive application. The time of successive switchovers tended to decrease. The reason for this behavior needs to be studied more closely (notice that the system should respond well to the first switchover rather than the second). Page faults are not suspected. The results presented here are from the last of a series of two or three successive switchovers.

The sets of files used by the benchmark jobs were organized in two ways: all four sets (16 files) in the same resilient group; and divided into two groups of two sets (8 files) each, with a different primary machine for each group. The latter eliminates the need to create agents on the backup during switchover.

The jobs were run for 4 minutes before performing the switchover. No major differences were observed between cases where the application is remote or local to the primary. The results are shown in Table 3.

Table 3. Switchover performance

Number of applications	Switchover time (seconds)	
	All files in same group†	Files divided into two groups‡
0	0.45	NA
1	1	NA
5	2.5	NA
7	3.3	NA
10	5	2.2
20	10.7	4

† Switchover includes the time to submit all agents and re-open 16 files per application on the backup machine.

‡ Agents on the backup machine exist prior to switchover. The switchover includes the time to re-open 8 files per application on the backup.

7 Summary

The methodology and design of a cluster that provides highly available data have been presented.

The project demonstrates that high availability can be transparent. Neither the application nor the database need to be changed.

Data is replicated at a primary and one or more backup machines. Data operations are intercepted by a stub that performs remote data access at the primary. Mechanisms have been developed to preserve data integrity and to perform seamless switchover upon primary failure. Consistent cluster state is maintained by a combination of protocols for global serialization and asynchronous dissemination of events. The entire cluster is resilient to single failures.

We have implemented an experimental prototype cluster of IBM AS/400 machines. Tests have been conducted using a commercial database application, without any change to the application or the database. The prototype substantiates the feasibility of our approach.

Acknowledgments

The authors wish to thank Jim Ranweiler for his invaluable contributions and support.

References

[1] Amir Y., D. Dolev, S. Kramer and D. Malki, "Transis: A Communication Sub-System for High Availability," *Proceedings of the 22nd International Conference on Fault Tolerant Computing*, pp. 76-84, 1992.

[2] Bartlett J.F., "A Nonstop Kernel," *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 22-29, 1981.

[3] Bhide A. and S.P. Morgan, "A Highly Available Network File Server", IBM Research Report RC16161, 1990.

[4] Birman K.P. and R. Van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, 1994.

[5] Borghoff U.M., *Catalogue of Distributed File/Operating Systems*, Springer-Verlag, 1992.

[6] Cristian F., "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems," *Distributed Computing*, vol. 4, no. 4, pp. 175-187, April 1991.

[7] Cristian F., H. Aghili, R. Strong and D. Dolev, "Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement," *Proceedings of the 15th International Conference on Fault Tolerant Computing*, pp. 200-206, 1985. To appear in Information and Computation.

[8] Cristian F., B. Dancey and J. Dehn, "Fault Tolerance in the Advanced Automation System," *Proceedings of the 20th International Conference on Fault Tolerant Computing*, pp. 6-17, 1990.

[9] Finkelstein S.J., "Algorithms and System Design in the Highly Available Systems Project," *Fault-Tolerant Distributed Computing, Lecture Notes in Computer Science no. 448*, pp. 138-146, Springer-Verlag, 1987.

[10] Gray J., *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufman Publishers Inc., 1991.

[11] Griefer A. and R. Strong, "DCF: Distributed Communication with Fault Tolerance," *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pp. 18-27, 1988.

[12] Halpern J., B. Simons, R. Strong and D. Dolev, "Fault-Tolerant Clock Synchronization," *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pp. 89-102, 1984. To appear in Journal of the ACM.

[13] IBM Corporation, IBM Application System/400 Technology Journal (Version 2), Publication No. S325-6020-00, 1992.

[14] Kaashoek M.F. and A.S. Tanenbaum, "Group Communication in the Amoeba Distributed Operating System," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 222-230, 1991.

[15] Lindsay B.G., L.M. Haas, C. Mohan, P.F. Wilms and R.A. Yost, "Computation and Communication in R*: a Distributed Database Manager," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 24-38, Feb. 1984.

[16] Liskov B., S. Ghemawat, R. Gruber, P. Johnson, L. Shrira and M. Williams, "Replication in the Harp File System," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 226-238, 1991.

[17] Marzullo K. and F. Schmuck, "Supplying High Availability with Standard Network File Systems," *Proceedings of the 4th International Conference on Distributed Computing Systems*, pp. 447-453, 1988.

[18] Melliar-Smith P.M., L.E. Moser and V. Agrawala, "Broadcast Protocols for Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 17-25, Jan. 1990.

[19] Peterson L.L., N.C. Buchholz and R.D. Schlichting, "Preserving and Using Context Information in Interprocess Communication," *ACM Transactions on Computer Systems*, vol. 7, no. 3, pp. 217-246, Aug. 1989.

[20] Satyanarayanan M., J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, vol. C-39, no. 4, pp. 447-459, April 1990.

[21] Siegel A., K. Birman and K. Marzullo, "Deceit: A Flexible Distributed File System", Technical Report 89-1042, Department of Computer Science, Cornell University, Nov. 1989.