A Reliable Storage Management Layer for Distributed Information Retrieval Systems

Charles L. A. Clarke
Philip L. Tilker
Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Allen Quoc-Luan Tran
Computer Science
University of Toronto
Toronto, Ontario, Canada

Kevin Harris
Antonio S. Cheng
Electrical & Computer Eng.
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

We present a storage management layer that facilitates the implementation of parallel information retrieval systems, and related applications, on networks of workstations. The storage management layer automates the process of adding and removing nodes, and implements a dispersed mirroring strategy to improve reliability. When nodes are added and removed, the document collection managed by the system is redistributed for load balancing purposes. The use of dispersed mirroring minimizes the impact of node failures and system modifications on query performance.

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Information Storage and Retrieval—*Systems and Software*

General Terms

Measurement, Performance, Experimentation

Keywords

Cluster Computing, Self-Managing Systems, Distributed Information Retrieval

1. INTRODUCTION

Information retrieval systems are usually straightforward to parallelize. The standard approach is to split a collection into subcollections that are indexed on different nodes in a network of workstations [12, 19, 21]. Queries are sent to all workstations in the network, which perform independent searches on the subcollections. The results of these independent searches, which are small in size and require little bandwidth to communicate, are gathered centrally and merged to create the final result set.

A distributed information retrieval system based on a network-of-workstations architecture scales easily. As necessary, extra workstations can be added to improve query execution time, or to increase the size of the collection without compromising it, essentially producing a linear speedup with the number of nodes [12].

Unfortunately, the use of a network-of-workstations architecture produces corresponding reliability, availability, and system management problems. Assuming that node failures are independent and that time to failure is exponentially distributed, the mean-time-to-failure (MTTF) of the system as a whole decreases in proportion to the inverse of the number of nodes it contains.

Load balancing is essential for system performance. When a node failures or a new node is added to the system, the collection must be redistributed across the system to balance the load. Since a query is sent to all nodes and all nodes must respond before the results can be returned to the client, the slowest node to respond becomes a bottleneck. Rebalancing after nodes are added or removed must be automatic. Manual rebalancing is tedious and is unlikely to be performed consistently.

The system should provide high availability for query processing. Since the information retrieval application may be used interactively, it should take a second or less for the system to recover and continue query processing after a node failure.

This paper describes the architecture of a distributed information retrieval system that addresses these reliability, availability and system management issues, providing both redundancy and automatic load balancing. After a node failure, the system ensures that query processing can continue with minimal delay. We have taken a layered approach to the creation of the system. To the greatest extent possible we isolate the details of the application —-the services it provides, its internal file structures and algorithms — from the details of the software components that implement the storage management layer. The result, a *reliable storage management layer*, provides a framework for developing a broad class of information retrieval and digital

CIKM'03, November 3–8, 2003, New Orleans, Louisiana, USA.

library applications. With the addition of a protocol-conversion layer, existing single-node applications may be converted into reliable distributed applications. Applications written to the specification outlined in Section 4 can be extended to a network-of-workstations environment using the baseline architecture described in Section 2 and immediately become compatible with the storage management layer, acquiring its benefits with minimal additional effort.

2. BASELINE ARCHITECTURE

Before discussing the details of the storage management layer, we describe a baseline architecture for a distributed information retrieval system. The baseline architecture is a generalization of the architecture of the MultiText system [7], but represents a reasonable plan for parallelizing many information retrieval applications. The baseline architecture makes no provision for node failure or for the addition of new nodes. The storage management layer is targeted at this baseline architecture, improving reliability and simplifying system management, while having minimal impact on its components.

Figure 1 give an overview of the baseline architecture. Each node in the system runs a copy of the application, which stores and indexes its subcollection in a database on local physical storage. Two additional components tie the applications together into a single federated system. The *Query Marshaller/Dispatcher* accepts connections from external clients, distributes the clients' search requests to the applications for execution over their subcollections, merges the results and returns them to the clients. The *Update Marshaller/Dispatcher* serves a similar role for updates to the collection. To the external clients, the collection of single-node applications running on the workstation cluster appear to be a single large instance of the single-node application.

Load balancing during updates is a major responsibility of the Update Marshaller/Dispatcher. For simplicity, we assume that reasonable load balancing can be achieved by assigning each new document to the smallest subcollection. Of course, unusual patterns of deletes may leave the system in an arbitrarily unbalanced state, but this is unlikely to occur under normal usage and in any case the Update Marshaller/Dispatcher could, in principle, perform rebalancing by moving documents from one subcollection to another. Neither marshaller/dispatcher maintains state in persistent storage, and as a result, can execute on any node in the system, either a node where a copy of the application is also running or on a dedicated node.

It is worth examining the update model in some detail, since it is here that we place the greatest restrictions on the application. First, given our focus on information retrieval, we restrict updates to the addition and deletion of complete documents, or groups of related documents, along with their embedded images and other auxiliary data. Document edits require a deletion of the original in conjunction with the addition of a modified version. Second, we assume that documents

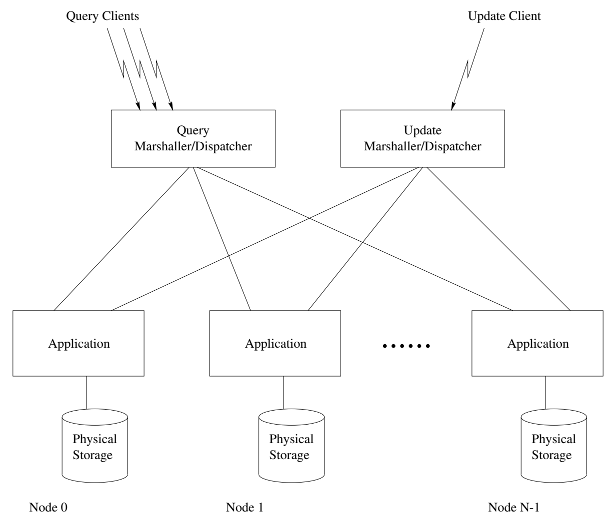

Figure 1: System architecture before addition of the storage management layer

may be added or removed at any time, but we assume that these updates are applied by a single external update client, which applies the updates sequentially. The alternative, supporting many clients concurrently updating the system is not generally considered in the context of IR systems, although it is the usual assumption for relational database systems and has been the subject of exhaustive study in that context. Third, we assume that updates may be grouped into batches, where the application of an update batch is a relatively long running operation. Finally, we assume that updates are not time critical and that it is acceptable to rollback and restart an update in the event of a failure.

Our update assumptions are similar to those generally made by others when addressing the problem of update in information retrieval systems. Most work in the area uses a simple document addition/deletion or append-only model in which updates are applied in batches to improve performance [2, 5, 10, 22]. One exception is Kamath and Ramamritham [15] who use an extended update model that permits multiple concurrent updates, groups updates into transactions, and avoids rollback in the event of a failure.

We require the system to support the standard ACID properties of transactions (Atomicity, Consistency, Isolation, Durability). Updates are grouped into transactions, which are applied atomically. After a transaction is committed, the changes are reflected in persistent storage and will survive a system shutdown or soft crash. Since the system is distributed, the application must be capable of participating in a two-phase commit.

In contrast to updates, we place few restrictions on the query interface. It must not be possible to perform updates through the query interface, and the application must respect the isolation property of transactions. Otherwise, the developer is free to design the query interface according to the needs of the application.

3. STRATEGY

The storage management layer handles node failure, the addition of new nodes, and the removal of existing nodes for administrative purposes. Consider the failure of a single node in Figure 1. This failure can be of one of two types. After a *soft crash*, the failed node can be restarted and reintegrated back into the system, although any ongoing query or update transactions running on the node are aborted and restarting the system may require several minutes. Any updates that have been committed are durable, and survive the crash. On the other hand, after a *hard crash* any data stored on the node is lost, and the failed node cannot be reintegrated into the system in its pre-crash state. The storage management layer ensures that sufficient redundancy is present in the system to survive the hard crash of a single node. In addition to node failures, the storage management strategy supports the addition of new nodes, rebalancing the system to take advantage of the additional resources. A planned removal of an existing node is treated as a hard crash (the node can simply be shut down or disconnected). To facilitate the performance of its responsibilities, the storage management layer also coordinates all external updates, subsuming the role of the Update Marshaller/Dispatcher.

The storage management layer uses timers and heartbeat messages broadcast periodically by each node to detect hard and soft crashes. Each node broadcasts a heartbeat message every $T_{broadcast}$ seconds. Other nodes use this heartbeat message to judge the node's status. As long as regular heartbeat messages are received, the node is judged to be *Alive*. If no heartbeat message is received from a node within a timeout period, T_{soft}, it is assumed that a soft crash of the node has occurred and the node is judged to be *Comatose* until a heartbeat message is again received. If a heartbeat message not received from a node for a second timeout period, T_{hard}, the node is judged to be *Critical*. Once a node is judged to be *Critical*, state information broadcast from other nodes is used to judge that a hard crash has occurred [23]. Presently, we use values $T_{broadcast} = 0.25s$, $T_{soft} = 1.0s$ and $T_{hard} = 500s$, giving a node sufficient time to restart before a hard crash is assumed. A new node asks to be added to the system by broadcasting heartbeat messages. When these messages are detected by existing nodes, the process of adding the node is initiated. Section 7 outlines the distributed consensus and group membership protocol, the *Global State Protocol*, used to supervise the addition and removal of nodes.

We use mirroring to cope with single-node failures. Before discussing our approach, we consider a simpler mirroring strategy in which the subcollection stored on one node is fully replicated on another. After a soft crash, the node where the replicated data is stored must immediately begin servicing queries on the replicated data. Otherwise, an unacceptable delay of several minutes in responding to queries may occur while the failed node restarts. This approach has a significant performance drawbacks when failures take place. Query execution time on the node containing the replicated data will roughly double, in turn doubling overall query execution time, since the slowest node in the system forms a bottleneck. After a hard crash or a planned node removal, the replicated data must be redistributed among the remaining nodes to rebalance the load. Given network bandwidth limitations, the time to required update file structures, data volumes in the tens of gigabytes, and the need to continue query processing, this rebalancing might require minutes or hours. Until the rebalancing has been completed, the impaired performance of the node servicing queries on the replicated data continues.

To avoid the load balancing problems inherent in this simple single-node mirroring strategy, we use a *dispersed mirroring* strategy. If there are N nodes in the system, the subcollection stored on node i is divided into $N-1$ *subsubcollections* of approximately equal size, which are stored on the others nodes: 0, 1,..$i-1$,$i+1$,...$N-1$. After a node failure, each of the remaining nodes immediately begins to service queries on the subsubcollection replicated from the failed node. If the node failure is a hard crash, each node absorbs the subsubcollection into its own subcollection, dividing and replicating it as it is absorbed. Under this dispersed mirroring strategy, a node failure increases the load on each remaining node by a rough factor of $1/N-1$. This increase occurs as soon as a soft failure is detected; no further rebalancing is needed.

The process is reversed when a new node is added to the system. Each node selects $1/N+1$ of its subcollection to form part of the subcollection stored on the new node, removes these documents from its subcollection, and chooses a different $1/N+1$ of its subcollection to be replicated on the new node, removing the existing replicas of these documents stored on other nodes.

3.1 Related Work

Self-management and fault tolerance are key elements of "autonomic" or "recovery-oriented" computing [4]. Verma et al. [24] describe an autonomic architecture to automate the mirroring and replication of client-server applications. Recognizing that manual replication and mirroring is both cumbersome and failure-prone, they propose a system that dynamically creates and manages replicas of application components in a distributed environment.

Replication techniques have long been used in relational database systems to improve both performance and availability [13]. In the context of digital libraries, Crespo and Garcia-Molina [8] have proposed a layered architecture for replicating documents over independent sites for long-term archival storage. Lu and McKinley [16] discuss the use of partial replication to improve the performance of IR systems. Dispersed mirroring was used in the Tiger video server [3] for reasons similar to our own.

Fox et al. [11] describe the implementation of HotBot. HotBot was based on a cluster architecture, with documents statically assigned to nodes. Node failures were tolerated through fast restart and by allowing the size of the collection to degrade when necessary. Data was not replicated. In Inktomi, a predecessor to HotBot, each database partition was accessible to several nodes, which could relieve each other in the event of failure.

Coded redundancy is an alternative to mirroring that imposes substantially less storage overhead. A simple parity scheme, such as that used in level 5 RAID systems [6], increases the per node storage requirements only by a factor of $1/N$ in a system containing N nodes.

The use of coded redundancy can impose a substantial performance overhead on queries after a node failure and on updates at all times. After a node failure, an access to data stored on the failed node requires a reconstruction that accesses data stored on all remaining nodes. Similarly, a disk write during an update requires the computation of coded redundancy and a write to local disk on all nodes. The impact on performance takes the form of reduced network and disk bandwidth, an increased computational load, and the latency of disk accesses across the network. Given the potentially negative performance impact, and given that disk storage costs little more than a dollar per gigabyte, we have chosen not to pursue these coded redundancy techniques for our system.

4. APPLICATION ENVIRONMENT

The application developer creates a single-node application. The developer is given as much freedom as possible in the design of this application and in the facilities of the operating environment that can be used in its implementation. To extend this single-node application to a function in an distributed environment, the developer writes only the Query Marshaller/Dispatcher. The storage management layer provides a standard Update Marshaller/Dispatcher and implements the storage management strategy.

The cooperation of the application is required to realize our dispersed mirroring strategy. For queries to be applied to a subsubcollection after a node failure it must be indexed. After a hard crash, a subsubcollection must be absorbed into the subcollection on each node and be further divided for replication. When a new node is added to the system each subcollection must be divided to make its contribution to the new node.

Figure 2 illustrates the interfaces required of the application by the storage management layer. The application takes the form of a server, accessed over a network by its clients using a connection-oriented protocol (TCP/IP in the current implementation).

The application must support two types of clients. *Query clients* use the search and retrieval services provided by the application. The application must be capable of supporting multiple query clients, each with its own separate connection. The application must also support a connection from a single *update client*. The update client is responsible for interacting with the application to add and delete data. The protocol used by the query client is defined entirely by the requirements of the application. On the other hand, the protocol used by the update client is precisely specified by the storage management layer.

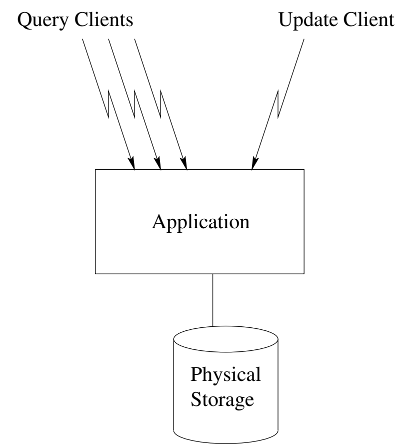

Figure 2: Application environment

5. UPDATE PROTOCOL

The update protocol is outlined in Figure 3. The basic update commands `addDocument` and `deleteDocument` add and remove documents from the subcollection managed by the application. Each document has a unique identifier assigned to it by the update client, and this identifier is sent as a parameter to the `addDocument` command, along with the size of the document and the actual document itself. The `deleteDocument` command takes a document identifier as its sole parameter. Both `addDocument` and `deleteDocument` return a *capacity* value indicating the percent free space remaining in the database after the command completes. The `getCapacity` command returns this value without modifying the database.

The `import` and `export` commands simplify the transfer of documents from one database to another. A separate network connection is established to make the transfer, permitting the transfer format to be specified entirely by the application. For example, a portion of an index file structure could be extracted from one database and loaded into another with the index maintained in a convenient format and without the update client being aware of the details of this format. When an `export` command is initiated, the update client passes a list of document identifiers to the application and the application returns the network address (TCP/IP port number) where the data corresponding to these documents is available. The `import` command takes a network address (TCP/IP host and port number) as a parameter and establishes a network connection to transfer the data.

The `begin`, `prepare`, `commit` and `abort` commands group the previous commands into transactions. A transaction starts with a `begin` and ends with either a `commit` or `abort`. The `prepare` command is used during a two-phase commit. When a `prepare` command is received by the application, it logs its commit readiness in persistent storage and acknowledges the command. No further update commands are accepted until the transac-

`addDocument`	add document to the subcollection
`deleteDocument`	remove document from the subcollection
`getCapacity`	return percent free storage remaining
`import`	import data in internal format
`export`	export data in internal format
`begin`	begin transaction
`prepare`	prepare to commit or abort transaction
`commit`	commit transaction
`abort`	abort transaction
`resize`	extend or truncate the size of persistent storage
`getSize`	return current, maximum and minimum size of persistent storage

Figure 3: Application update protocol

tion is committed or aborted. If a soft crash occurs after the `prepare` has been acknowledged by the application, the first command sent by the update client after restart should be a `commit` or `abort`. An unexpected `commit` or `abort` is ignored after a restart.

The storage management layer also places limits on the application's use of persistent storage. The remaining two commands listed in Figure 3, the `resize` command and the `getSize` command instruct the application in its use of this storage.

6. STORAGE MANAGEMENT LAYER

Figure 4 presents the software components implementing the storage management layer on one node of the system. All of these components run on each node of the system. Most of the components appearing in the figure are separate processes, communicating via stream connections and shared memory. The application may itself consist of multiple processes, at the discretion of the application developer.

Multiple instances of the application run on each node. If the systems contains N nodes, each node runs N copies of the application. One copy of the application, the *primary*, manages the subcollection stored on the local node. The remaining $N-1$ copies of the application, the *secondaries*, each manage a subsubcollection replicated from a different node. Each copy of the application interacts directly with the Query Marshaller/Dispatcher, but receives all update commands through an instance of the Application Update Proxy.

6.1 Global State Machine

The Global State Machine (GSM) broadcasts and receives heartbeat messages and uses these messages to implement the Global State Protocol, a distributed consensus and group membership protocol, outlined in Section 7. The Global State Protocol detects node failures, supervises the addition and removal of nodes, and coordinates the process of committing external updates. The steps in these activities that require global coordination are reflected by state changes in the GSM. Other software components monitor these state changes through local connections to the GSM.

6.2 Application Update Proxy

The Application Update Proxy serves two major roles. First, it maintains a list of documents managed by the application, along with their sizes. This list is used to guide the division and merging that takes place during the addition or removal of a node. Second, the proxy contacts the appropriate source of update commands and acts as an intermediary between this source and the application. In the case of the primary, the source of updates is the local Node Controller. In the case of the secondaries, the source of updates is normally the remote Node Controller associated with the replicated subsubcollection, but in the event of the hard crash the source of updates becomes the local Node Controller. If a soft crash of the remote node is detected through the GSM, the proxy aborts any update operations in progress, closes any network connection to the remote Node Controller and waits for the state of the remote node to change. If a hard crash is detected, the proxy contacts the local Node Controller and acts on its commands.

6.3 Node Controller

The Node Controller is responsible for implementing the actual addition and removal of nodes and the application of external updates. The Node Controller monitors the GSM and is contacted by the Update Marshaller/Dispatcher, the local primary proxy, and its $N-1$ remote secondary proxies. Under normal operation, when no nodes failures have been detected and a new node is not being added, the Node Controller receives update commands from the Update Marshaller/Dispatcher and applies these changes to the local subcollection via the primary proxy and to its replicated subsubcollections via the remote secondaries. The final commit or abort of these updates is coordinated by the GSM.

We examine node removal in detail. When a hard crash is detected, the Node Controller aborts any current update operation and waits to be contacted by the associated local secondary proxy. It then exports all data from the local secondary and imports it into its local primary and remote secondaries. If the storage capacity of the primary or any of the secondaries would be exceeded, the Node Controller aborts the operation and the system enters a special state where the missing node must be replaced. The Node Controller then instructs the primary and secondaries to prepare to commit the changes and waits for acknowledgment. Once acknowledgment is received, the Node Controller logs its commit readiness persistently and informs the GSM.

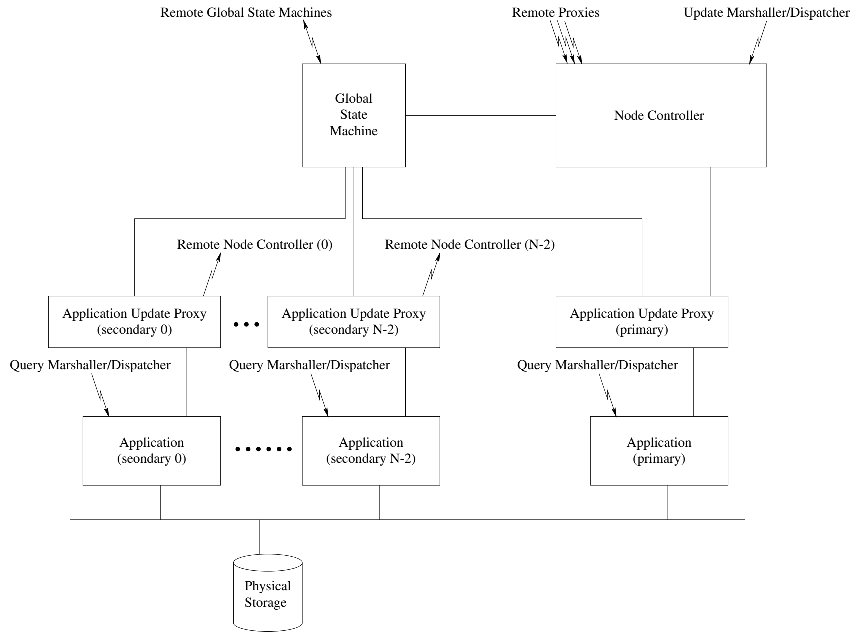

Figure 4: Node architecture

The GSM coordinates the final step of the two-phase commit. As well as the Node Controllers, the Application Update Proxies directly monitor the GSM to determine if the operation is committed or aborted. Direct monitoring permits a secondary proxy to proceed even if a soft crash of its associated remote node occurs. If removal of the node is committed, the Node Controller shuts down the local secondary associated with the removed node and releases the persistent storage allocated to it.

As a final step in the rebalancing process, the Node Controller resizes the physical storage allocated for its remote secondaries using the `getSize` and `resize` commands. A soft crash could occur before this resize completes. Whenever an proxy establishes a connection with a Node Controller, the Node Controller immediately checks the size of its physical storage and resizes it if necessary.

To recover from a soft crash, the failed node checks its commit readiness in its log and compares it with the current GSM state. If the local node has been removed from the system, it reinitializes its persistent storage and initiates an add/replace node operation through the GSM. Otherwise, it runs its primary and secondary proxy and application servers, and waits for contact from its remote secondaries, resizing them if necessary.

6.4 Marshaller/Dispatchers

Neither the Query Marshaller/Dispatcher nor the Update Marshaller/Dispatcher maintain state in persistent storage. As a result, if the node on which a marshaller/ dispatcher is running fails, a new marshaller/dispatcher can be created on any node in the system. Furthermore, since the applications can support multiple query clients, multiple copies of the Query Marshaller/Dispatcher can be running in the system at any one time.

As part of its operation, the Global State Protocol, discussed in Section 7, designates a *leader* node and a *crown prince* node. Our strategy is to run the Query Marshaller/ Dispatcher on both the leader and crown prince and the Update Marshaller/Dispatcher on the leader. If the leader fails, the Query Marshaller/Dispatcher on the crown prince is used by the external clients, while the Update Marshaller/ Dispatcher is not required until the leader recovers or is removed from the system. When the leader is removed from the system, the Global State Protocol promotes the crown prince, and the Update Marshaller/Dispatcher is run on the new leader.

7. GLOBAL STATE PROTOCOL

The Global State Protocol provides distributed consensus and group membership services for the system, managing the membership of nodes that form the system, providing notification of soft crashes, removing nodes that have hard crashed, detecting new nodes and incorporating them into the system, and coordinating updates to the collection [23]. This section briefly outlines the protocol, which is a variant of standard consensus and group membership protocols [1, 17], but is unusual in its support of both soft and hard crashes.

The Global State Protocol is implemented by the GSM on each node. Each GSM maintains a *state* value for its own node, and a *state vector* containing either the last state value received from every other node in the system or an indication that node is *Comatose* or *Critical*. The local state value and state vector are used to form part of an *alive matrix* that records the last state vector received from each node.

7.1 Heartbeats

The GSM is responsible for broadcasting the heartbeat message every $T_{broadcast}$ seconds and receiving heartbeat messages from other nodes. Heartbeat messages include the current state vector for the node, and as heartbeat messages are received from other nodes, the GSM updates its state vector and alive matrix.

In addition to the state value, each node maintains and broadcasts a *modification number*, a *last commit number* and a *commit flag*. The modification sequence number is incremented after each attempted system modification (node addition, removal or replacement; collection update) and the last commit number is incremented after each successful modification. The commit flag indicates the outcome of the last modification (commit or abort). These values are used during crash recovery, permitting a node to complete a two-phase commit and to determine if it should rejoin the system or reinitialize itself and initiate an add/replace.

Our broadcast model closely follows the *Timed Asynchronous Model* [9] and is easily implemented by using standard Ethernet broadcast, for example. Broadcast need not be reliable, but messages broadcast from one node must be delivered in order at other nodes. Broadcast messages may be lost because of transient network problems, such as congestion or buffer overflows, or because of a heavy load on a node, but the probability of delivery should reasonably high. Otherwise, values for T_{soft} and T_{hard} would have to be unacceptably long. While soft crash designations are revoked as soon a heartbeat message is received, and an incorrect soft crash designations can be tolerated, a hard crash designation is irrevocable. A predicate computed over the alive matrix is used to minimize the probability that inconsistent hard crash designations will be made on different nodes.

If a hard crash occurs, we assume that another hard crash will not occur until the failed node has been removed or replaced. Under this one-fault model, a subsequent hard crash would imply a system failure. The protocol requires that node failures follow a *failfast* model, where the node simply stops upon failure, as opposed to a *Byzantine* model, where the node may act in an arbitrary, potentially vicious, manner.

7.2 Protocol Overview

Every system modification undergoes a two-stage process in order to maintain system consistency. Every member moves through these two stages together to ensure a primary partition: a single agreed view of the system among all the members of the system. State changes in the GSM on each node are used coordinate movement through the stages. The first, or initiation, stage is required to decide and agree on what modification to perform. The second stage is required to determine each member's ability to commit the modification.

In this dynamic system, there may be several modification "requests" at any one time. For example, a failure may be detected while a new node is requesting addition or while the Update Marshaller/Dispatcher wishes to update the collection. However, no two members should attempt to modify the system in two different directions. The initiation stage is required to keep the members of the system consistently making the same modification.

The GSM accomplishes the initiation problem by implicitly designating a leader for the system, which is responsible for initiating any modifications. It is important for each member to be able to locally determine the correct unique leader of the system. Leader determination is implemented by assigning a unique index to each member of the system when it is added. Every node maintains this list, and the leader is the member with the smallest index.

To handle the case that the leader itself has failed, a *crown prince* [14] is designated whose responsibility is limited only to the removal of the leader. As with the leader, it is necessary to locally determine a unique crown prince, and the node with the second smallest index takes this role. Upon successfully removing the leader, the crown prince itself becomes the leader and another becomes the crown prince (assuming more than one node still exists).

After the leader (or crown prince) has initiated a modification, the local clients of the GSM are notified and control is passed to the clients (specifically to the Node Controller) to make the necessary arrangements to complete the modification. The GSM waits for a response from the client before proceeding to the commit stage.

During the commit stage, the system as a whole must collectively decide on whether to commit or abort the modification. We use a decentralized two-phase commit which takes advantage of the periodic broadcast nature of the heartbeats. A node distributes its readiness to commit and collects the readiness of other nodes through the broadcast of GSM state changes. Normally the agreement of all N nodes is require to commit changes. However, if a hard crash is detected during a system modification, the system can proceed with the agreement of $N-1$ nodes. After completion of the commit stage, the failed node will be removed.

8. IMPLEMENTATION

We have implemented the storage management layer on a cluster computing facility in the School of Computer Science at the University of Waterloo. The cluster is built using off-the-shelf PCs and Ethernet hardware and runs a version of the Linux operating system. The implementation uses UDP Broadcast and TCP for communication between nodes.

Performance of the storage management layer depends heavily on the performance of the application it is supporting. The time required to complete a system modification — the addition or removal of a node or an update to the collection — depends to a large extent on the time required by the application to access and modify its own internal data structures.

To directly measure the overhead introduced by the storage management layer, we created a *null application*, which implements the protocol of Section 5, but does not perform any storage or retrieval functions. This null application does not maintain persistent data struc-

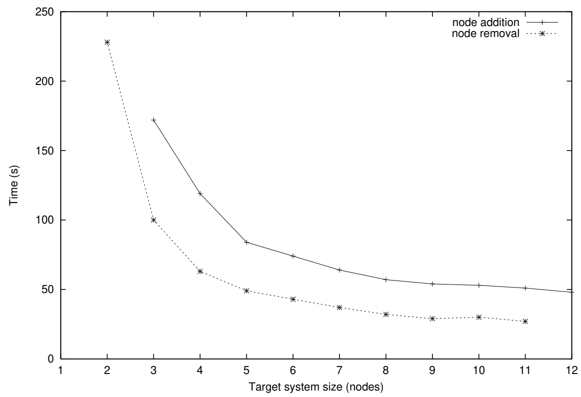

Figure 5: Node addition and removal overhead.

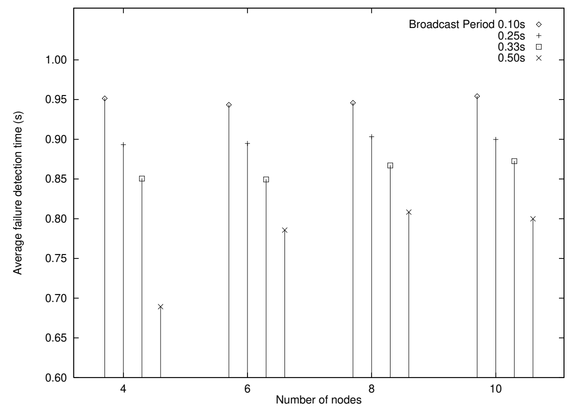

Figure 6: Average soft crash detection time

tures. The contents of any documents added to the application are discarded. The `import` and `export` commands establish remote links to other copies of the application, but do not transfer data.

Nonetheless, from the point of view of the storage layer the null application is a functioning information retrieval application. The application proxies maintain their own file structures recording the documents stored in each instance of the application. The node controllers supervise data transfer from instance to instance as the document collection is updated, and nodes are added and removed.

We evaluated the overhead using a testbed of 12 nodes and a collection of 2,000,000 documents. To begin, an initial system of two nodes was created, containing all 2,000,000 documents, and then new nodes were sequentially introduced up to a maximum of 12 nodes. At each step, we measured the time from the introduction of a new node to the commit of the associated add operation. Once the maximum of 12 nodes had been reached, nodes were then sequentially removed until only 2 nodes remained. During the removal process, we measured the time from the point that the leader recognizes the failure to the point where the leader commits the remove operation. The time to recognize the failure was not included, since it is dominated by on the value of T_{hard}. This process was repeated three times. The average of the observed measurements is reported in Figure 5.

The time required to perform an add or remove operation is greatest when there are only two nodes in the system. As the number of nodes increases, the time required decreases, as the average number of documents per node drops.

The performance of the storage management layer is particularly important in the case that a soft crash occurs. An immediate response is required to allow query processing to continue. To evaluate performance in the event of a soft crash, we measured the time taken for clients of the GSM on all nodes to be notified of a soft crash and to response to it. We measured the response times for various broadcast periods ($T_{broadcast}$) and for systems containing different numbers of nodes. A value of $T_{soft} = 1.0s$ was used in all cases. The results are presented in Figure 6.

A soft crash is detected on individual nodes in time less than or equal to T_{soft}. Since the failure may occur at any point in the broadcast cycle, the actual time may be as little as $T_{soft} - T_{broadcast}$. This dependence on $T_{broadcast}$ is responsible for the drop in average detection time as $T_{broadcast}$ is increased.

9. CONCLUSION

We have described a layered architecture for improving reliability and simplifying the system management of a distributed information retrieval system implemented on a network of workstations. The layer supervises the addition and removal of nodes, moving data to implement load balancing and replicating data to prevent loss in the event of a single node failure. The architecture uses a dispersed mirroring strategy to minimize the impact of a node failure on query performance. While the application is required to implement and use specific interfaces, these interfaces are kept simple and general. This layered approach represents a compromise between tight integration of the storage management layer with the application, which would reduce its generality, and a completely transparent storage management layer, which would make our dispersed mirroring strategy difficult to implement. The cooperation of the application is limited to a simple update protocol that can be implemented as a layer above an existing application.

The work could be extended in a number of ways. The current approach requires updates to be batched and serialized through a single update client, and it cannot handle the failure of more than one node. Support for multiple concurrent updates and system modifications involving multiple nodes should be considered. We currently assume a homogeneous environment, but if a cluster grows over a period of time it is likely to contain different generations of hardware. Implementing a load balancing policy that is aware of the performance characteristics of the application and adjusts the load balancing strategy to account for hardware differences would be a significant challenge.

While the application interface is sufficient to meet the needs of many of information retrieval and digital library applications, there are situations in which extensions are needed or for which the overall approach is not appropriate. To meet the requirements of applications that store streaming multimedia data, the system would need to provide real-time performance guarantees.

The approach presupposes a baseline architecture consisting of multiple instances of a single-node application federated to form a distributed application. Applications whose architectures violate this assumption may be difficult or impossible to support. For example, many distributed information retrieval applications generate and maintain collection-wide information that must be updated as documents are added and removed [18]. For applications utilizing an inverted index, an alternative to partitioning a collection by document is to create a single global index, assigning a subset of the index terms to each node [20]. While a global index can outperform document partitioning, other strategies must be developed to address the issues of reliability and system management in this context.

10. REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control And Recovery In Database Systems.* Addison-Wesley, 1987.

[2] E. Bertino, B. C. OOI, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems.* Kluwer, 1997.

[3] W. J. Bolosky, J. S. Barrera, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. The tiger video fileserver. In *6th International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1996.

[4] A. B. Brown and D. A. Patterson. Embracing failure: A case for recovery-oriented computing (ROC). In *2001 High Performance Transaction Processing Symposium*, October 2001.

[5] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *20th VLDB Conference*, pages 192–202, September 1994.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[7] G. V. Cormack, C. L. A. Clarke, C. R. Palmer, and S. S. L. To. Passage-based query refinement. *Information Processing and Management*, 36(1):133–153, 2000.

[8] A. Crespo and H. Garcia-Molina. Archival storage for digital libraries. In *Digital Libraries '98*, pages 69–78, 1998.

[9] F. Cristian and C. Fetzer. The timed asynchronous system model. Technical Report CS97-519, University of California, San Diego, January 1997.

[10] D. Cutting and J. Pedersen. Optimizations for dynamic inverted list maintenance. In *13th ACM SIGIR*, 1990.

[11] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *16th ACM Symposium on Operating System Principles*, pages 78–91, October 1997.

[12] D. Hawking, N. Craswell, and P. Thistlewaite. Overview of the TREC-7 very large collection track. In *Seventh Text REtrieval Conference*, November 1998.

[13] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. *Replication Techniques in Distributed Systems.* Kluver, 1996.

[14] F. Jahaniana, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design, and implementation. In *12th Symposium on Reliable Distributed Systems*, 1993.

[15] M. Kamath and K. Ramamritham. Efficient transaction support for dynamic information retrieval systems. In *19th ACM SIGIR*, August 1996.

[16] Z. Lu and K. S. McKinley. Partial collection replication for information retrieval. *Information Retrieval*, 6(2):159–198, April 2003.

[17] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[18] A. L. Powell, J. C. French, J. Callan, M. Connell, and C. L. Viles. The impact of database selection on distributed searching. In *23rd ACM SIGIR*, pages 232–239, August 2000.

[19] B. Ribeiro-Neto, E. S. Moura, and M. S. Neubert. Efficient distributed algorithms to build inverted files. In *22nd ACM SIGIR*, pages 105–112, August 1999.

[20] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Digital Libraries '98*, pages 182–190, 1998.

[21] A. Tomasic and H. Garcia-Molina. Performance issues in distributed shared-nothing information retrieval systems. *Information Processing and Management*, 32(6):647–665, 1996.

[22] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *ACM SIGMOD International Conference on Management of Data*, pages 289–300, May 1994.

[23] A. Q.-L. Tran. A network management facility for a fault-tolerant distributed information retrieval system. Master's thesis, University of Toronto, 2000.

[24] D. C. Verma, S. Sahu, S. Calo, A. Shaikh, I. Chang, and A. Acharya. SRIRAM: A scalable resilient autonomic mesh. *IBM Systems Journal*, 42(1):19–28, 2003.