A Reliable Storage Management Layer for a Distributed Information Retrieval System

by

Philip Tilker

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Computer Science

Waterloo, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Philip Tilker

I authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Philip Tilker

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 IR Systems

An information retrieval (IR) system is used for performing searches on extremely large document collections. Before an IR system is able to answer client queries, the document collection must first be stored in a way that allows for efficient querying. Indexing a document collection involves building all necessary data structures to allow a client to efficiently query the entire collection. The type of data structures used depend entirely on the type of queries to be performed. Typically, some form of an inverted list is used in an IR application [21, 7, 18]. Google is an example of an IR system in which the document collection being searched consists primarily of web pages in the Internet.

The above description of an IR system could also be applied to any database system. Both systems index large collections and both permit multiple concurrent clients to perform queries over the document collection. The main different between a database system and an IR system is the type of information being indexed. Databases typically require a schema describing the data to be stored. Once a schema has been provided, all data must match the schema. It is this structure that differentiates database systems form IR systems. IR systems index unstructured data. In the case of Google, the unstructured data is a web page. Web pages come in many different shapes and sizes. Some web pages such as corporate web pages may be very structures, whereas personal web pages me lack structure all together.

Databases must support both queries and updates. Consequently, the data structures used in a database system must allow for efficient updates in addition to efficient queries. In an IR system, updates might not be supported. The update model of an IR system is such that it no updates may be allowed. In the case where updates are permitted, updates may be append-only, or, it may provide for the addition/deletion of whole documents. Additionally, document updates are applied in batches to improve performance [1, 4, 9, 18].

This is in contrast to the online update model that most database systems must support.

1.2 Distributed IR Systems

Converting a centralized IR system into a distributed IR system based on a cluster of workstations is typically a straightforward process. The document collection is evenly divided into sub-collections, with each sub-collection indexed one node in the distributed system [13, 15, 17]. Client queries are solved by sending the query to each node. The individual results of the client queries from each node are then centrally gathered, combined and transmitted back to the client. Figure 1.1 shows a distributed IR system with 4 nodes.

There are many benefits to a distributed IR system based on a cluster of workstations. A distributed system based on a cluster of workstations scales easily. New nodes can be added to increase the storage capacity of the distributed system as a whole. Secondly, query response time can be improved. Query processing time can be decreased linearly as the number of nodes in the system grows [13].

While query response time can be linearly improved with the addition of new nodes to the distributed system, this benefit can be realized only if the work to be performed by each node when executing a client query is the same. Since all nodes must finish their work before a client query is fully answered, the slowest node in the system will dominate the query performance. As a result, load balancing is a crucial requirement of any distributed system. The document collection must be redistributed every time a node is added to, or removed from, the distributed system. Without any tools, this task is a manual, non-trivial task that is unlikely to be performed consistently.

Although there are definite advantages with distributed IR systems, these benefits come at the cost of system reliability and system management. 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 [6]. A self-managing system is desireable to provide faster response to specific events (node addition/node failure) as well as reduce human error. Brown and Pattersion describe the need for "autonomic" or "recovery-oriented" systems from the context of Internet-connected transactions in order to increase the availability of services [3]. The autonomic architecture described by Verma et al. [20] automates the replication of client-server applications.

The contribution of this work is the design and implementation of a general software layer for distributed information retrieval systems. Our software layer, the RDDS (Redundant Data Storage System) address the system reliability and system management issues that exist in current distributed IR systems. The RDDS provides data redundancy to ensure the entire document collection remains available in the presence of a single node failure. Additionally, automatic load balancing is performed whenever the document collection is updated, or when the nodes participating in the distributed system change. In the event of a node failure, the RDSS allows query processing to continue with minimal delay.

The RDSS system is designed in two distinct layers. The bottom layer is the information retrieval application itself. This layer has no concept of participating in a distributed system and views itself as the entire information retrieval system. The second layer is a management layer that coordinates the efforts of many application instances to create a reliable distributed storage system. The management layer makes few assumptions or requirements on the underlying application and can therefore be applied in many different situations.

1.3 RDSS Goals

The RDSS is a software management layer that facilitates the development of distributed IR applications. Whether the IR application is new, or if the RDSS is being applied to a previously existing distributed IR system, the RDSS alleviates many burdens of a distributed system. Unfortunately, the RDSS is not able to solve these issues without some restrictions on the underlying application interface. For the most part, the application interface is defined entirely by the application itself, however, some restrictions are necessary. These restrictions are detailed in chapter 4. The real benefit of the RDSS is that given a previously existing distributed IR system, with minimal effort, the RDSS can be integrated into the distributed system to provide the redundancy and load balancing features.

The RDSS improves the system reliability of a distributed system by accounting for the fact that the MTTF of the distributed system decreases as the system grows. As the number of nodes participating in the distributed system grows, the MTTF between node failures decreases linearly with the number of nodes participating in the system. A node failure poses the danger of causing a temporary loss of data availability. In more severe circumstances, permanent data loss may occur as a result of a node failure. Under a RDSS managed distributed system, fault tolerance is introduced to solve these two problems. In the event of a single node failure, no permanent data loss will occur, even if the permanent storage containing the data of the failed node is physically damaged. Additionally, the same fault tolerance designed to avoid permanent data loss is activated immediately to avoid any temporary loss of data availability.

As a distributed system grows, so too does the amount of time spent managing the system. Specifically, as nodes are added to, and removed from, the distributed system, data must be transferred between nodes for proper load balancing. The overhead in calculating an even workload between all nodes increases as the distributed system grows. Additionally, a mechanism is needed to detect failed nodes in a timely manner and to then take the necessary corrective steps. All of these tasks are time consuming and can be non-trivial, raising the possibility of human error to occur. In a RDSS distributed system, the load balancing aspects of the distributed system are automated, as is the detection of failed nodes. Since the RDSS is self-managing, no human intervention is necessary when a node failure occurs. When a node is added to, or removed from, the distributed system, the RDSS will detect the new, or lost node, and will automatically take appropriate action to redistribute the entire document collection over the new set of nodes participating in the distributed system.

1.4 Strategy

The strategy used in the RDSS is to introduce fault tolerance to each node. In addition to indexing a sub-collection of the entire document collection, each node will also index pieces of the sub-collections belonging to other nodes. If a node fails, these pieces can be activated to any avoid loss of data. As nodes are added to and removed from the distributed system, the RDSS automatically detects changes in the distributed group membership and redistributes the document collection to maintain the fault tolerance and to make efficient use of all resources.

The document update model is limited to two types of operations. Whole Documents can be *added* to, or *deleted* from a document collection. Document updates are implemented by first deleting the old copy and then adding the new copy of the document as a single transaction.

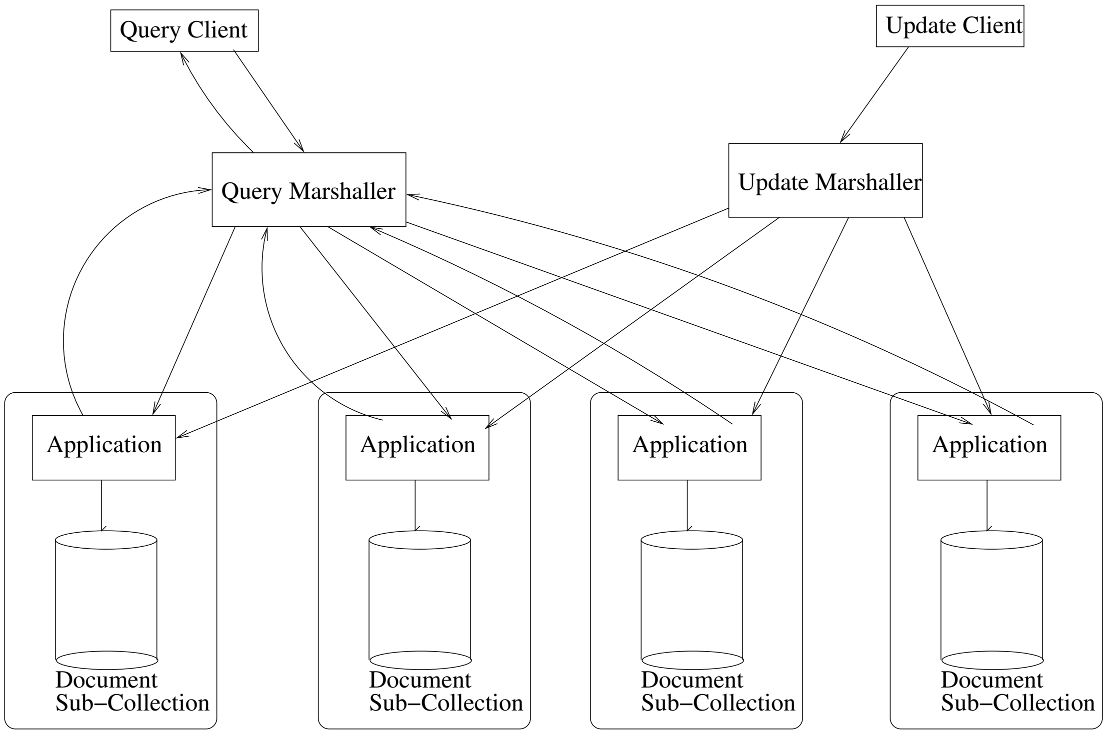

Figure 1.1: Distributed IR System

Chapter 2

Background and Related Work

2.1 Failures

The failure of an object can be viewed generally as the failure of the object to perform its desired operation. In computing terms, there are two general types of failures: software and hardware.

A piece of software can be written correctly, yet designed poorly, and as a result of the design, the software may fail due to its inability to address all possible scenarios. An example of poor design could be banking software not accounting for negative balances. Alternatively, a piece of software may be designed flawlessly, yet implemented poorly, and its implementation becomes the cause of the failure (segmentation faults). A hardware failure occurs when a physical piece of equipment stops responding correctly. Hard drive failures are a common failure of a computer. It is difficult to describe the effect of a hardware failure on the software components running on the node. Clearly, any software component interacting with the failed hardware is affected. However, if the software component is designed to handle failures, there may be no impact. On the other hand, there could be serious side effects which could lead to the failure of all software components.

For the purposes of this thesis, we are focussed on hardware failures or maintenance changes, rather than software. We assume that all relevant software is designed and implemented correctly. Therefore, during our discussions of various components, it is impossible for one component to have failed on the node while the others remain correct, functioning components. As we will see in chapter 3, our definition of a hardware failure relates to the inability of a node to communicate with every other node in the system. When a node has failed, we assume that all software components on that node have also failed. Since this assumption may not always be true, we can assume it is nonetheless because the GSM algorithm described in section 2.3 provides a mechanism for ignoring a node deemed to have failed should that node still be able to communicate with the rest of the distributed system.

Finally, we make one further assumption regarding failures. We assume that all failures are fail-stop, as opposed to byzantine failures [11, 10]. The nature of byzantine failures can make it difficult to event detect a failure. The assumption of fail-stop failures allows us to assume that any process that fails is unable to deceive the distributed system by pretending to act as a correct process would, but not performing the same activities as a correct process.

2.2 Distributed Consensus

In any distributed environment, it is sometimes necessary for a distributed consensus to be reached. It turns out that achieving a distributed consensus can be difficult, if not impossible, depending on the situation. In a time-free, asychronous distributed system, reaching a distributed consensus is possible only when failures are not allowed [11]. The general theme of the proof is that a correct process is unable to determine if a process has failed, or if it is merely slow. It is this inability that makes services like distributed consensus impossible in a time-free asynchronous distributed system [10]. Fortunately, all computers manufactured today come equipped with high-precision clocks [8]. Cristian and Fetzer formally define a *timed asynchronous* distributed system model and show that this model accuratey describeds current distributed systems. They go on to prove that services such as distributed consensus can be solved in a timed asynchronous distributed system [8].

2.3 GSM Algorithm

Before the potential of any distributed system based on a cluster of workstations can be fully realized, some well known problems with distributed systems need to be addressed. In the RDSS, we address two specific problems. First, we need some mechanism for establishing and maintaining a distributed group membership. As nodes are added to, and removed from, the distributed group, we must always be able to definitively determine which nodes are currently participating in the distributed group. Secondly, a facility to support distributed transactions must exist in order to guarantee the ACID properties that our update model requires. The global state machine (GSM) was designed to solve these two problems. In this section, we outline the concepts and details of the GSM algorithm which are assumed throughout this thesis. The GSM was developed by Tran and Clarke [19]. Our description of the GSM is only a summary of the work by Tran and Clarke.

2.3.1 Overview

The GSM is a combination of a finite state machine and a distributed algorithm. On each node in a distributed system, there exists one GSM. Each GSM maintains its own local view of what state the distributed group is currently in. Through a distributed algorithm, all GSMs in the distributed group reconcile any differences between individual local views to come up with one, agreed upon, global view of the distributed group. The state maintained by each GSM describing the distributed system consists of several pieces of information. The first piece of information maintained is the set of nodes comprising the distributed group membership. In addition to the distributed group membership, each GSM stores the last time it heard from each remote GSM in the distributed group. Information not pertaining to group membership includes the current state of the GSM, as well as how many group operations have been attempted and how many of those operations were successful. The latter information is used to support the atomic commit facility offered by the GSM. The state of a GSM indicates what operation the GSM believes the distributed group is currently performing, or about to perform. Examples of states could be to add a new node to the distributed group membership or to abort the current operation being attempted. Details of all states are given in section 2.3.3.

In order for all GSMs in the distributed group to come up with a global view of the distributed system, periodically, each GSM sends its local view to all remote GSMs. We call the transmission of the local view of a GSM a heartbeat since any GSM receiving a local view from a remote GSM has reason to believe the remote node is alive. Each heartbeat message contains the most recent local view of the GSM sending the heart beat message. Upon receiving a heartbeat message from a remote GSM, the receiving GSM compares its local view of the distributed system to that of the remote GSM and reconciles any differences. Any changes made to the local view of the receiving GSM as a result of the reconciliation process will be made known to all other GSMs the next time the receiving GSM sends a new heart beat message. The constant sending and receiving of heart beat messages between GSMs in the distributed system allows the GSMs to continually monitor each other's health.

2.3.2 Distributed Consensus

The distributed consensus problem is key to the success of the GSM. For any consistency to be present, all GSMs in the distributed group must be able to agree on a common decision before any operation can occur. At any given time, it is possible that a new node is requesting to be added to the distributed group, while an old node has failed and should be removed. Furthermore, an external client might want to make an update to the distributed system. In addition to all of these possibilities, the two phase commit protocol also requires a distributed consensus among all participating group members. Only one operation can be performed at a time and so an algorithm is needed for all members of the distributed group to come to a distributed consensus.

To avoid the problem of multiple operations being initiated at once, the GSM algorithm allows for only one specific member of the distributed group to initiate an operation. Within the distributed group membership, one node is chosen to be the *king* node and it is this node that is responsible for initiating all group operations. The king node initiates an operation by making a transition to a `CONSENT` state. This state transition is the clue to all other GSMs that an operation is being initiated. The next time the king node sends its heartbeat message to all remote GSMs, the remote GSMs will see the `CONSENT` state and at this point, the distributed algorithm of the GSM come to a consensus regarding whether the operation should be performed.

The GSM algorithm guarantees correct operation even in the presence of a node failure. When a node fails, the king node initiates the removal of the failed node from the distributed group membership. This policy is not sufficient because if the king node fails, there is no node to initiate its removal. In this special situation, the *crown prince* node steps in and initiates the removal of the king node. The crown prince node is just like any other node except that it has the ability to initiate the removal of the king; no other operations can be initiated by the crown prince.

The selection process of the king node and the crown prince node requires some form of a distributed consensus. One solution is for a central node to arbitrarily choose a king and crown prince and then notify all other GSMs in the distributed group. This type of centralized consensus is not resilient since a single failure of the central node could halt any progress. A distributed consensus is ideal since the GSM algorithm provides a mechanism for a distributed consensus to occur. Unfortunately, the GSM algorithm requires a king node to already exist in order for a distributed consensus to occur.

To avoid the pitfalls of both of these two proposed solutions, the GSM algorithm instead designates the king node and the crown prince node as opposed to electing them. Every node in the distributed group membership is assigned a unique number corresponding to the order in which nodes were granted membership in the distributed group. The first node to enter the distributed group membership is assigned value 0, the second, value 1 with the n^{th} node to enter the distributed group assigned the value $n - 1$. Since all GSMs agree on one global view of the distributed system, the numbers assigned to each node are the same across all GSMs. As a result, the king node is chosen to be the node whose value is 0 and the crown prince is chosen to be the node whose value is 1. This selection process is guaranteed to select the same two nodes in all GSMs running in the distributed group.

There is still a fundamental problem with the above algorithm. If the king node and the crown prince node are unable to communicate with each other, the king node might try to remove the crown prince at the same time as the crown prince tries to remove the king node. To make matters even worse, a network partition might leave the system in a state where half the nodes can see the king node and not the crown prince and the other half can see the crown prince and not the king. To help solve situations like the ones above, additional information is needed. In general, the king or crown prince will not initiate an operation unless all nodes are in agreement. No progress can occur until all nodes agree on a view of the system. In the case where a network partition occurs, no progress will be made because consensus is needed by all but one node. This consensus can not occur unless the partition is such that only one node is secluded from the rest. In that case, the secluded node will appear to have failed and it will be removed since the remaining nodes will see each other and agree to remove the secluded node.

Once established, the GSM algorithm is able to add nodes, remove nodes, and perform document collection updates. The remaining detail is how a distributed group membership is first established in a consistent manner. When creating a new distributed membership, the first two nodes of the group are manually specified. The node whose IP address is smallest becomes node 0 and the other node 1. Following the rules set above, the king node and crown prince node can now be uniquely identified and the GSM algorithm can now proceed without intervention.

2.3.3 Finite Automata

The state of the GSM contains the operation the GSM is attempting to perform. All operations must begin in the `STABLE` state and follow the transitions as shown in figure 2.1.

The actual number of states is much more than that shown in figure 2.1. We have simplified the state transitions by removing the operation being performed. There are, in fact, three `CONSENT` states, three `PREPARE`, three `COMMIT` and three `ABORT` states corresponding to the update operation, the node addition operation and the node deletion operation. The reason for the duplicated states is that each of the `CONSENT` states specifies precisely what operation is being initiated and the same holds true for the other states. Figure 2.2 shows the actual states used when adding a new node to the distributed group membership.

Finally, there are a few additional states not mentioned here that are used only when creating a new distributed group, when a node is restarting or when a new node is available to be added to the currently existing group membership. While these states are important to the proper functioning of the GSM, they are not important from the perspective of the RDSS. In fact, the only states of interest from the perspective of the RDSS are the

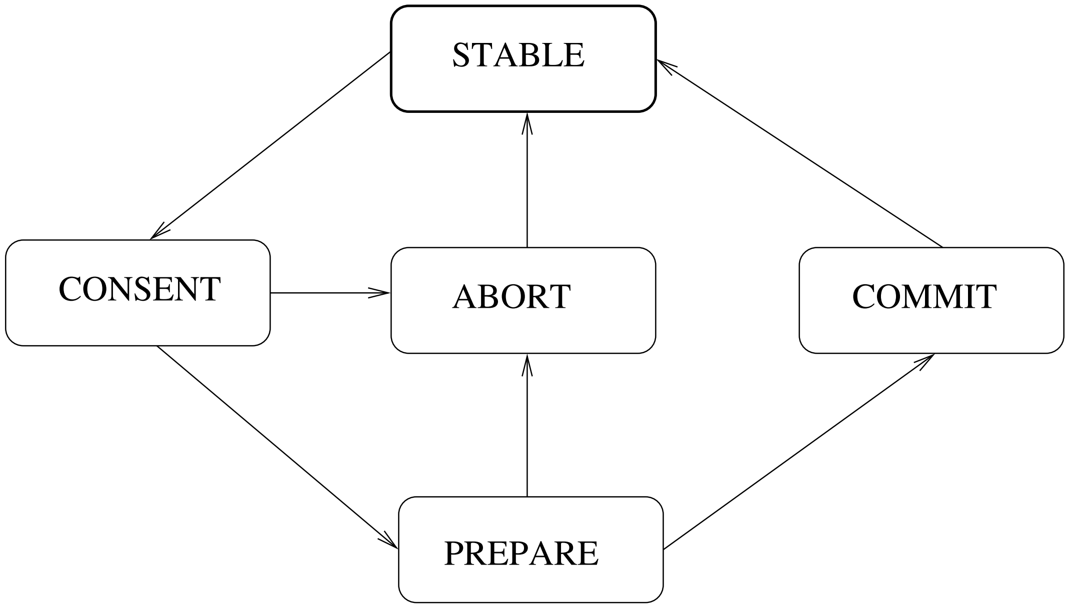

Figure 2.1: GSM State Transitions

PREPARE and the STABLE states along with the remaining information contained in a heart beat message.

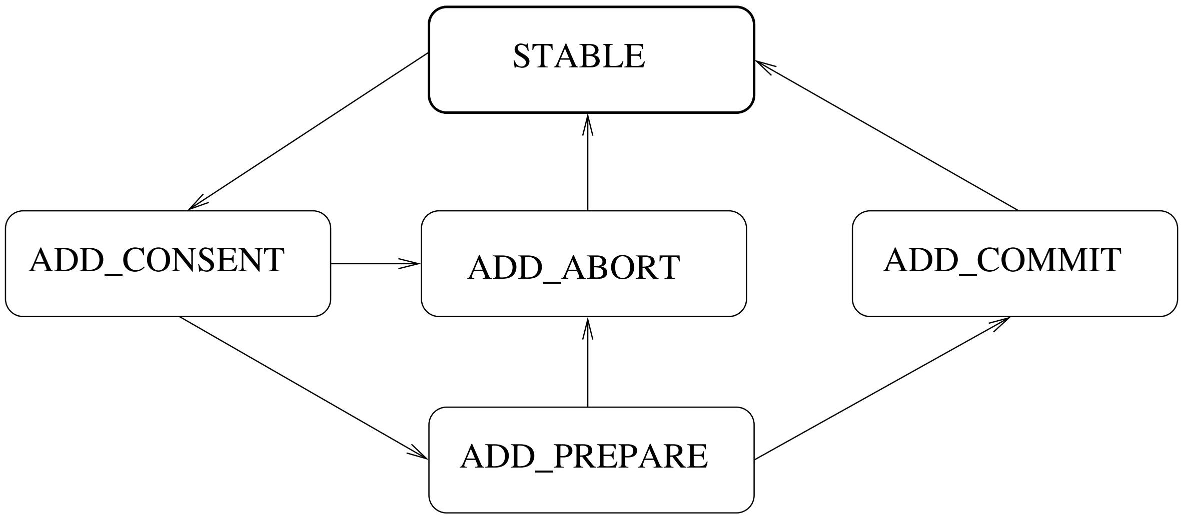

Figure 2.2: GSM State Transitions

2.4 Baseline Architecture

Before discussing the design of the RDSS, we first present the baseline architecture for which the RDSS was targeted. The application for which we target is a typical information retrieval application. While we impose no requirements on the query model of the application, the application must allow multiple concurrent client connections to query the document collection being indexed by the application. The protocol for client queries is specified entirely by the application itself.

The architecture presented is that of the MultiText [7] system; however, this system is a reasonable approach to parallelizing information retrieval systems and as such, can be used as a guide for many different types of information retrieval applications. The update model is a batch update model. Furthermore, the only two document collection operations are to add a complete document and to delete a complete document. The query model is interactive requiring an extremely fast response time.

The application must also allow for updates to be made to the document collection being stored by the application. Although the query model for the application is defined entirely by the application itself, the update model is not. The first restriction we impose is that update operations are limited to the removal and addition of complete documents. Document modifications are implemented by first removing the old document and then adding the new modified version of the document. The addition and removal of documents, must occur as a single transaction guaranteeing the ACID properties of transactions. Secondly, it is typical of information retrieval applications that the document collection remains relatively stable over time and as a result, update operations can be batched together into a single update operation. Because of this, the application is not required to support multiple concurrent update clients, although we place no requirement that multiple concurrent update clients be disallowed. The only implication of multiple concurrent update clients is the ACID properties of transactions. Finally, as a further result that updates occur infrequently, we assume that updates are not time sensitive and in the event of a failure, the update transaction can be aborted and restarted at a later time.

From the perspective of our update model, the ACID properties have the following implications. Atomicity requires any document collection changes being performed as part of a single update must all occur is a single unit. At no point in time shall the application allow only a subset of the changes to be seen by external clients. The consistency property has two main implications. Firstly, the application, when performing an update, must guarantee that any client queries must be executed over the previously document collection only. Secondly, when an update has been committed, the application must immediately stop executing queries over the old document collection and begin using the new collection. We leave it up to the application to decide what to do with client queries that cross of the boundary of a commit (those clients who began their query session before the commit operation that are still active after the commit operation). Several options are available such as shutting down the client, or delaying the commit of the update until all previously existing clients have finished. The durability requirement has no special implication with our update model. All that is required is that once an update is committed, the application is responsible for storing its document collection indefinitely. Finally, the requirement of isolation is trivialized by our update model. Since our update model does not allow multiple concurrent update clients, no special action is needed by an IR application to satisfy this requirement.

In the baseline architecture, each node participating in the distributed system runs a copy of the application, known as an application instance. The application is responsible for indexing the document sub-collection assigned to the node and for responding to client queries. Two additional components are needed to make all application instances appear as one single federated application instance. The *Query Marshaller* and the *Update Marshaller* are used to give a single point of contact for both client queries and update queries to the distributed system. To an external client, all application instances running on each node in the network of workstations appear as one large unified information retrieval system.

In the RDSS, external query clients contact the query marshaler and begin performing queries just as if the client were connected to an application instance directly. The query marshaler reads the client query to be executed and then distributes the query to each application instance in the system. Each application instance executes the query over its sub-collection and sends the results back to the query marshaler. The results from each node are gathered by the query marshaler and after all nodes have responded, the results are merged and the final merged result is sent back to the client. The result that is sent back to the client should be identical to the result that would be sent by a single-node application indexing all documents in the distributed system. A caveat of the role of the query marshaler is that the query marshaler must sufficiently understand the application query model in order to accurately be able to merge results.

The update marshaler performs an analogous role for update clients as the query marshaler does for query clients. In addition to distributing the updates to all nodes in the distributed system, the update marshaler has the very important role of load balancing. Load balancing is achieved by distributing the document updates according to a distribution policy.

Chapter 3

Design

In this section, we explain the design behind the RDSS. We begin by first introducing all components of the RDSS. In the subsections to follow, we elaborate and give more details on what each particular component does and why it is required. Throughout our discussion, we will talk from the perspective of a single node or a single component of a node. To help remove ambiguity when referring to different nodes, we use the terms *local* and *remote* nodes. In a distributed system with n nodes, from the perspective of any node, there is 1 local node and $n-1$ remote nodes. The local node is the node from whose perspective we are speaking while a remote node is any other node in the distributed system that is not the local node. When we refer to the local node of a component, we are referring to the node where the component is running.

3.1 System Overview

We begin with the baseline architecture described in chapter 1, section 1.1. To support the replication used within the RDSS, additional application instances are added. Replication is implemented by taking each application instance in the baseline architecture and further dividing its document sub-collection into document sub-sub-collections. The number of sub-sub-collections is dependent on the number of nodes in the system. In a system with n nodes, each application instance's sub-collection is divided into $n-1$ sub-sub-collections. A copy of the document sub-sub-collections is stored on each of the remaining nodes in the system. Figure 3.1 shows a distributed system with 4 nodes.

In the baseline architecture, each node contained only one instance of the application. However, in figure 3.1, there are four instances of the application on each node. Generally speaking, in a RDSS distributed system with n nodes, there are n application instances running on each node. One of these application instances is referred to as the *primary* application instance while the remaining $n-1$ application instances are referred

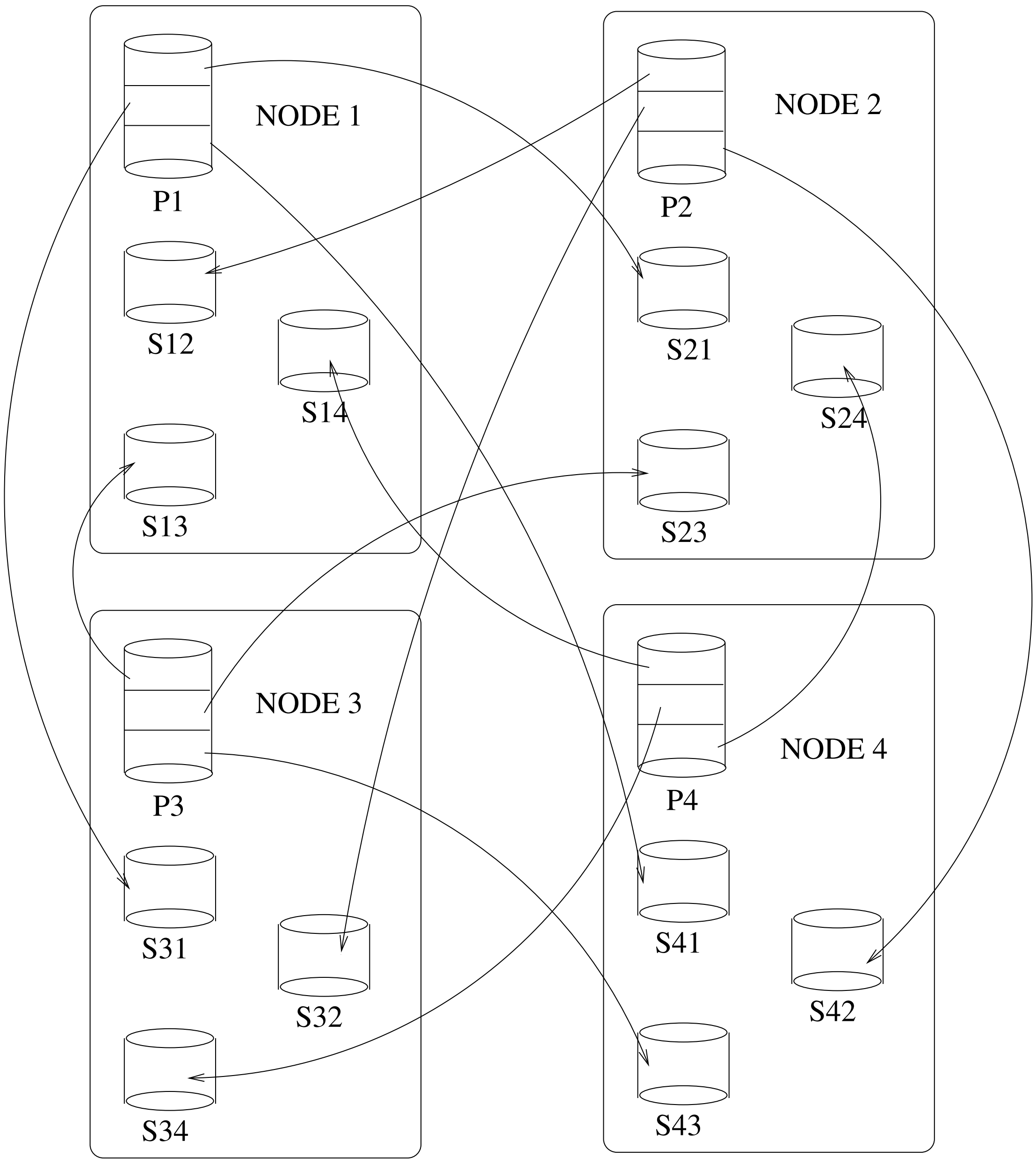

Figure 3.1: Application Instances in the RDSS

to as *secondary* application instances. The primary application instance is used for storing the document sub-collection assigned to the local node for indexing purposes. The secondary application instances are used for replication only. Each secondary application instance replicates a subset of the document sub-collection assigned to the primary application instance of a remote node. Referring back to figure 3.1, the application instances labelled $P1$, $P2$, $P3$, $P4$ are primary application instances and correspond to the application instances that existed in the baseline architecture. The application instances labelled $Sij, 1 \leq i, j, \leq 4, i \neq j$ are the secondary application instances. The figure also shows how the document sub-collections of each of the application instances are entirely replicated in the system.

For clarity, we denote the primary application instance running on node i as P_i. We denote the $n-1$ secondary application instances running on node i as $S_{ij}, 1 \leq j \leq n, j \neq i$. From the perspective of node i, node i is the local node, while nodes $j, 1 \leq j \leq n, j \neq i$ are the remote nodes. Furthermore, a *foreign application* instance is a secondary application instance running on the local node but is being used to replicate a subset of the data belonging to the primary application instance of a remote node. A *remote application* instance is a secondary application instance running on a remote node that is replicating data belonging to the primary application instance of the local node. The only difference between a foreign application instance and a remote application instance is perspective. Each secondary application instance is both a foreign application instance and a remote secondary application instance. From the perspective of node i, the $n-1$ foreign application instances are $S_{ij}, 1 \leq j \leq n, j \neq i$ and the $n-1$ remote application instances are $S_{ji}, 1 \leq j \leq n, i \neq j$. The secondary application instance S_{ij} is a foreign application instance from the perspective of node i and a remote application instance from the perspective of node j.

To facilitate the replication needs of the RDSS, several new components have been added to each node. In the baseline architecture, the only additional components were the query marshaler and the update marshaler. These two components still exist, although we have renamed the update marshaler. Table 3.1 lists all node components and a brief description of what each component does. More design details are given about each component in this chapter. Implementation details are given in chapter 4.

Component	Description ‘
GSM Controller	Run GSM Algorithm
IO Proxy	Facilitate inter-node communication
Four11 Server	Facilitate resource location
Node Controller	Manage a node during a group operation
Distribution Server	Distribute documents according a distribution policy
Application Proxy	Simplify Application Requirements
Application	Respond to client queries

Table 3.1: Node Components

3.2 Data Distribution Requirements

3.2.1 Overview

Although not fully in the scope of the work described in this thesis, we offer a brief discussion of data distribution schemes. A data distribution scheme is a policy used to decide where a document in the global document collection will be stored. In an IR system with only one application instance, the distribution policy is trivial; all documents must be indexed by the single application instance. When more than one application instance exists, the answer as to where a document should be indexed is not so obvious. Perhaps each application instance on a node is responsible for indexing documents of a particular type or genre. In this case, a classification of each document would be needed before it could be properly assigned. In distributed environments where there are multiple nodes, each possibly running multiple instances of the application instance, the distribution scheme could possibly become quite complex. The answer to what type of distribution scheme should be used depends primarily on the anticipated used by clients and any requirements imposed on the distributed system.

One possible data distribution scheme is to ensure each node in the distributed system is responsible for indexing roughly the same number of documents. This scheme leads to a balanced distribution of storage requirements (assuming roughly even document sizes) but does not guarantee an balanced load in terms of work. It is possible that for a given client query, all documents with results matching the query are located on a single node, even though the global document collection is evenly distributed across all nodes in the system. Conversely, another possible distribution policy could be to distribute all documents in the system such that the amount of work performed by each node is approximately equal. While such a distribution scheme is not as simple as the first, it could be accomplished by assigning an expected workload to each document and ensuring that the sum of all expected workloads is the same from node to node. The downside to this approach is that the storage requirements for each node may be wildly different. In addition to different workload and storage schemes, the use of heterogenous systems can allow for nodes to have varying capabilities in terms of processing power or storage capacity. These factors might also influence the policy used for distributing documents. In our prototype system, the distribution policy used is to maintain an even number of documents being indexed at each node in the system

3.2.2 Redundant Data Distribution Scheme

As the name suggests, the redundant data distributed scheme (RDDS) used in the RDSS uses redundancy to allow for node failures or removals. Specifically, we require a data distribution scheme that can tolerate a single node failure and still provide access to the entire document collection. Several designs have been proposed for fault-tolerant distributed systems [2, 12, 14, 5]. The RDDS described here is similar to that used in the Tiger Video Server [2]. Exactly two copies of the global document collection are stored in the distributed system. The *primary* copy is stored in the primary application instances while the *secondary* copy is stored in the secondary application instances. In a system with n nodes, there are n primary application instances and $n(n-1)$ secondary application instances. The RDDS makes no assumptions about what sort of distribution policy is used. All the RDDS does is define some minimum requirements that must be adhered to by any distribution policy. Before we define the RDDS, we introduce some notation. Let $D(P_i)$ be the set of documents stored by primary application instance P_i. Similarly, let $D(S_{ij})$ be the set of documents stored by the secondary application instance S_{ij}. Finally, let D_g denote the global document collection. The RDDS is defined in terms of four axioms that must always hold for any distribution. These axioms are shown in table 3.2.

1	$D_g = \bigcup_{i=1}^{n} D(P_i)$
2	$D(P_i) \cap D(P_k) = \emptyset, i \neq k$
3	$D(P_i) = \bigcup_{j=1}^{n} D(S_{ji}), i \neq j$
4	$\forall i, D(S_{ji}) \cap D(S_{ki}) = \emptyset, j \neq k$

Table 3.2: RDDS Definition

The first two requirements of the RDDS definition state that the global document collection must be divided into n, pairwise disjoint subsets. Each subset is assigned to one primary application instance for indexing. Furthermore, the union of all subsets much equal the global document collection. Similarly, requirements 3 and 4 state that each primary document sub-collection be divided into $n-1$, pairwise disjoint subsets. Each subset is assigned to one remote application instance for replication. Again, the union of all subsets must equal the document collection stored by the primary application instance. Any distribution of the global document collection satisfying the 4 conditions of the RDDS is said to be a *valid* RDDS distribution. The RDDS does not require a balanced distribution in order to function as designed. The only benefit provided by a balanced distribution is performance. Figure 3.2 shows a valid RDDS distribution of a global document collection containing 300 documents. For simplicity, documents are uniquely assigned an integer value from 1 to 300. It can be easily verified that the distribution shown satisfies all conditions of the RDDS. While this example is an extreme case where all documents in the global document collection are assigned to only one primary application instance, the RDSS will still function correctly because all four requirements of an RDDS are satisfied.

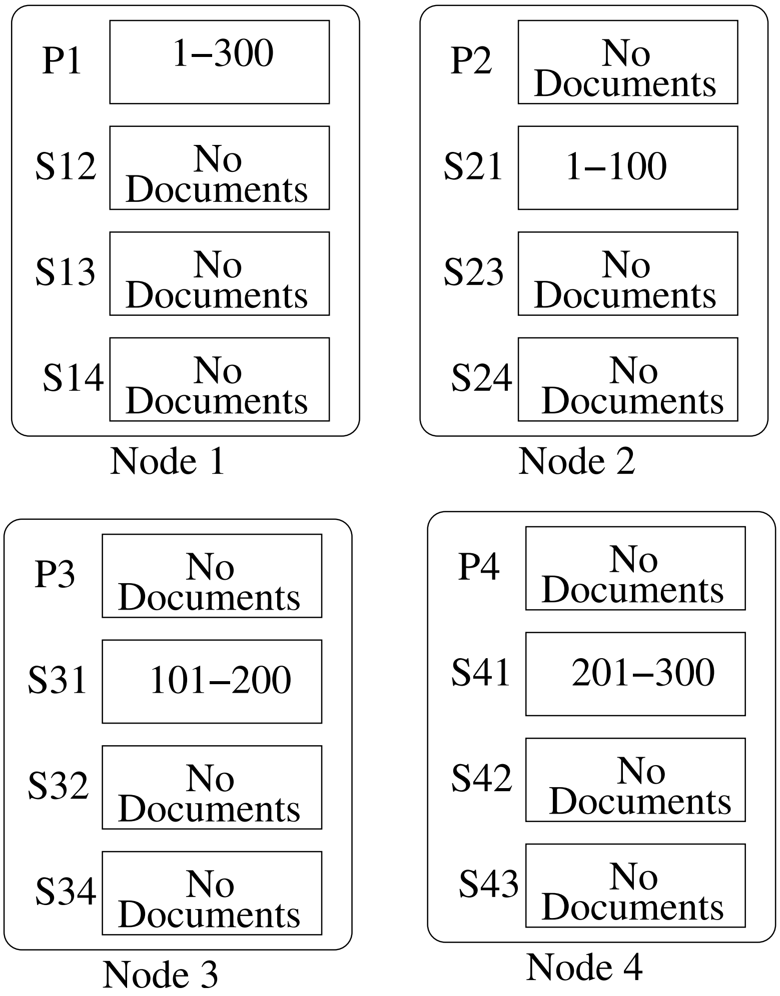

Figure 3.2: A Valid RDDS Distribution

3.2.3 Balanced Redundant Data Distribution Scheme

An balanced RDDS is a distribution scheme that satisfies the RDDS requirements that is also evenly distributed. A balanced distribution can take on many different meanings depending on the circumstances. Ideally, a balanced distribution is one in which, on average, all nodes in the system require roughly the same amount of time to answer a client query. If this is the case, no single node is the bottle neck of the distributed system and any redistribution of the documents would cause an increase in overall client query execution time.

Unfortunately, achieving a balanced distribution can be very difficult. One big obstacle to overcome is heterogenous systems. In heterogenous systems, the processing abilities and storage capacities of the nodes may be wildly different and thus require special consideration. Nodes that are slower than others may need to index fewer documents, as would nodes with smaller storage capacities.

A second obstacle, and one that is harder to define, is that of expected workload. The expected workload is the amount of work we expect a node to have to perform in response to client queries. The expected workload is dependent on many factors and may not be possible to compute. Let us assume a homogeneous system for the following discussion. For starters, it may be reasonable to expect that a node storing twice as many documents as another would have twice the expected workload. This assumption sounds reasonable assuming that all documents are roughly the same size. Varying document sizes complicate the process, but, can be overcome. Another complication could the nature of the client queries. Suppose we had a distributed system with 5 nodes, with each node storing all journals for a particular year. If a client performs a query for documents within a specific year, 4 of the 5 nodes will do no work while the last node performs all the work. Similarly, suppose we fixed our previous problem and had each node index journals for all 5 years. Now, if a client queries for journals relating to biology only, it is possible that all the biology journals were stored on single node and so the problem has not been solved.

Clearly, some of the examples above are contrived, but they nonetheless demonstrate some of the difficulties in achieving a balanced distribution of the document collection. In the last of the examples, we assumed a homogeneous system and so those same problems, combined with a heterogenous system become an even bigger problem to solve. In terms of evenly distributing documents based on storage requirements, this can be solved rather easily if all documents are the same size. However, if the documents are of different sizes, the ability to evenly distribute the documents based on storage requirements can be shown to solve the bin packing problem which is known to be NP complete. Distributing the document collection based on expected workload may require assigning an expected workload value to each document and then distributing the documents so the sum of each node's document's expected workload is similar. Again, this problem is similar to the bin packing problem.

Given the difficulties in evenly distributing a document collection, we have defined a balanced distribution to be a distribution in which each node is indexing approximately the same number of documents. If it turns out that the size of each document is roughly the same, this is a reasonable choice assuming that storage capacities of the nodes are sufficient. We now calculate the expected number of documents stored in each application instance assuming an even distribution.

We begin by assuming there are d documents in the global document collection and there are n nodes in the distributed system. Given these parameters, we know there are n primary application instances and $n(n-1)$ secondary application instances. We begin by dividing the global document collection into n disjoint subsets, each of size roughly d/n and assign each subset to one primary application instance. We further divide each document collection of a primary application instance into $n-1$ disjoint subsets, each of size roughly $(d/n)/n-1$. It is easy to show that our distribution scheme is a valid RDDS and it is also even by construction. Figure 3.3 is an example of an even balanced distribution of the same 300 documents shown in figure 3.2. For the distribution to be even, we would expect each primary application instance to store $d/4 = 300/4 = 75$ documents each. Furthermore, each secondary application instance should store roughly $(d/n)/n-1 = (300/4)/3 = 25$ documents each.

3.3 System Components

3.3.1 Application Proxy

The RDSS requires some additional bookkeeping on the part of the application in order to perform its tasks. While the application could be modified to do this bookkeeping, the RDSS has been designed so that minimal effort is needed for integration with a previously existing IR system. Consequently, it is desirable to minimize any changes that must be made to the IR application. Additionally, some of the bookkeeping would be the same regardless of the application. The application proxy was created to coexist with the application and to offload managerial tasks from the application itself.

With the exception of the application update protocol described later, no restrictions are placed on the application by the RDSS. However, certain pieces of information are necessary in order for the RDSS to perform its tasks. In our prototype system, the only information currently needed is a listing of all documents being stored by the application. Each document has an associated document header which contains pertinent information

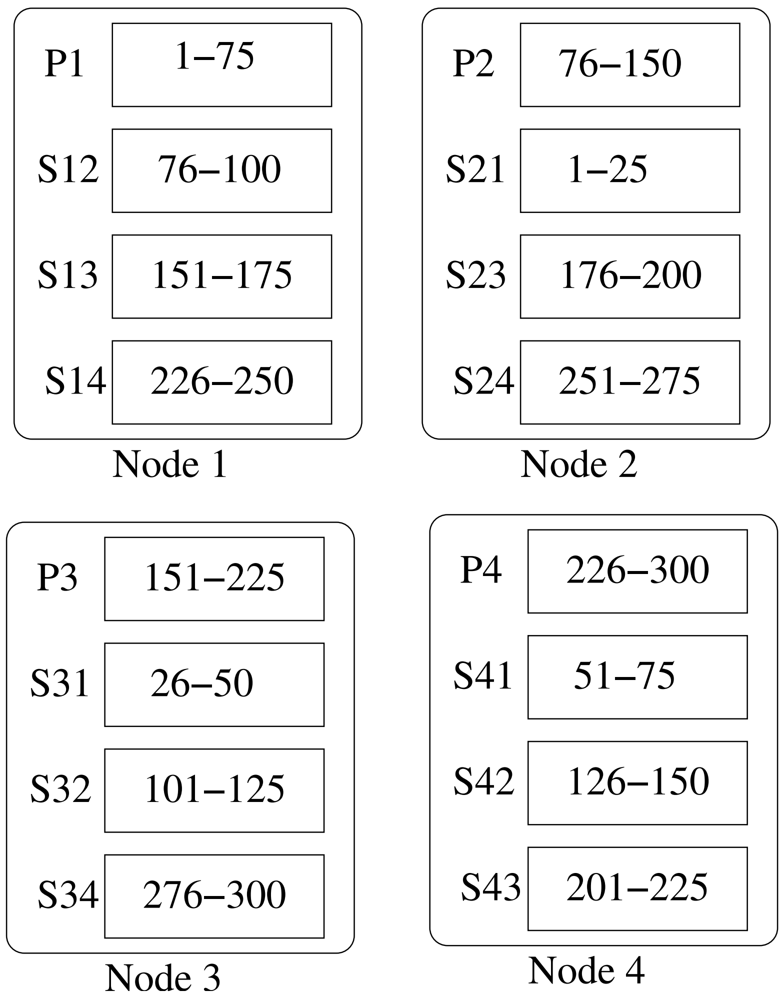

Figure 3.3: An Even, Valid RDDS Distribution

regarding a document, including the document identifier. A document identifier is a unique identifier that can be used to identifier a particular document in the global document collection. The application proxy is the component that maintains this information.

There is a one-to-one correspondence between application instances and application proxy instances. For each application instance created, an associated application proxy instance is also created. The application proxy maintains a list of document headers that correspond exactly to the set of documents stored in the associated application instance. Query clients never interact with the application proxy; they interact with the application or the query marshaler just as they did in the baseline architecture.

The application proxy must follow the entire update protocol just as the application must follow the update protocol. Any update commands must be sent to both the application and the application proxy so that the list of documents maintained by the application proxy corresponds with the documents stored in the application. Any update commands are no longer sent to the application directly. Instead, update commands are sent to the application proxy which then forwards them to the application. The flow of commands through the application proxy to the application allow both components to remain consistent in terms of the document list maintained by the application proxy and the documents stored by the application.

3.3.2 IO Proxy

In order to achieve the goal of quickly detecting and recovering from potential node failures within a few seconds, some form of fine granularity timeout control is needed for all inter-node communication. In the event that two components on different nodes are communicating and one of the nodes dies, the component on the remaining node could end up waiting indefinitely for data that will never arrive from the failed node. Without some higher level protocol, the remaining node has no way of discovering that the node it was communicating with has failed. Fortunately, some "higher level" protocol already exists. The GSM algorithm, via the GSM controller, maintains information regarding how long it has been since it last heard from each remote node in the system. By making use of this information, we can build a mechanism to quickly detect failed communications.

The solution used by the RDSS is to have a dedicated component on each node in the system that is solely responsible for all inter-node communication. This component is the IO Proxy. Suppose a component running on the local node wishes to establish a connection with a component running on a remote node. Let C_i denote a component running on node N_i and let IO_i be the IO Proxy running on node N_i. If C_1 wants to connect to C_2, the following steps take place. First C_1 establishes a connection with IO_1 using a well known TCP port. Once connected, C_1 tells IO_1 the IP address of node N_2 and the port number that C_2 is listening on. Next, IO_1 establishes a connection with IO_2 using the same well known TCP port, and again, the information regarding the IP address of N_2 and the port number of C_2 is relayed from IO_1 to IO_2. Finally, IO_2 establishes a connection with C_2. Instead of one direct connection from C_1 to C_2, one *virtual* connection between C_1 and C_2 has been established using three direct connections. Any data flowing between C_1 and C_2 must follow the path $C_1 \longleftrightarrow IO_1 \longleftrightarrow IO_2 \longleftrightarrow C_2$. A graphical representation of a virtual connection between two components is shown in figure 3.4.

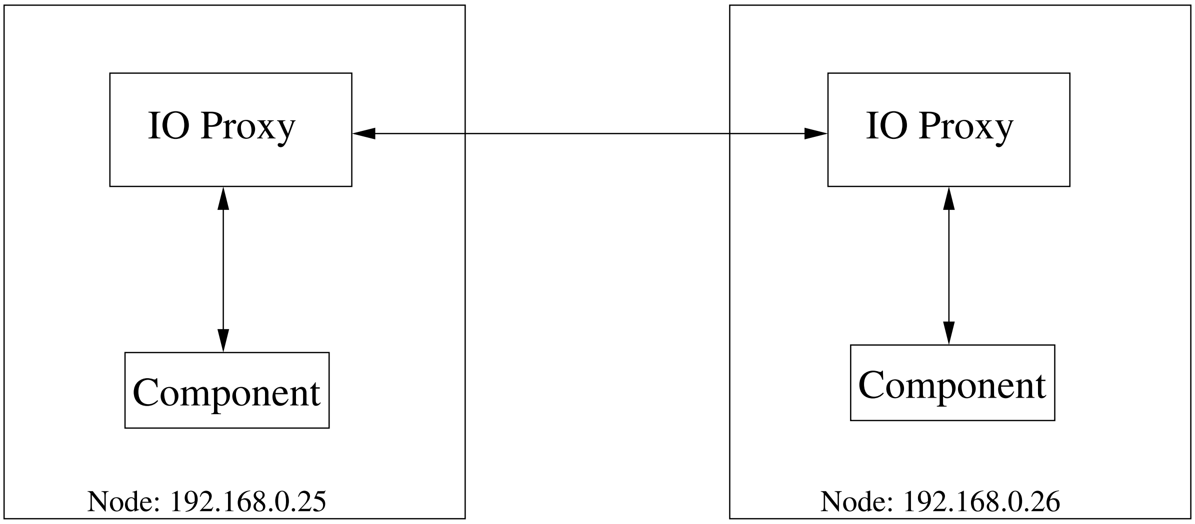

Figure 3.4: Inter-Node Communication

Since all communication flowing between nodes now goes through the IO Proxy on each node, the IO Proxy is in a position to sever the virtual connection at any moment. Severing a virtual connection is accomplished by closing the connection to the local component as well as the connection to the remote IO Proxy. When no errors occur within the distributed group, the IO Proxy will simply transfer data along the virtual connection. Eventually, the virtual connection will be closed by one, or both of the components ultimately communicating with one another. If, however, an error occurs within the system, the IO Proxy will sever all connections to the failed node to allow the system to discover the error and make forward progress.

In order to determine if an error has occurred in the distributed group, the IO Proxy continually monitors the state of its local GSM via its local GSM controller. The IO Proxy contacts its local GSM controller and tells the GSM controller that every heartbeat message generated should be sent to the IO Proxy in addition to being broadcast to all remote GSM controllers. If, after receiving a heartbeat message from the local GSM controller, the IO Proxy sees that the GSM is initiating the removal of a node, the IO Proxy assumes that the node has failed and so all communication with that node should be terminated. In this event, the IO Proxy will sever all virtual connections between any local components and any remote components on the failed node. This same process occurs in all IO Proxies in the system and so it is true that all virtual connections involving a failed node will indeed be closed.

Typically, the GSM controller will not decide that a node has failed for several minutes. However, we would like to recover from failed nodes quickly, and so it is not sufficient to simply wait for the GSM controller to finally decide that a node has indeed failed. Instead, the IO Proxy uses information contained in the heartbeat messages to decide when a *potential* node failure may have occurred. Recall that each heartbeat message includes the amount of time since the local GSM has received a heartbeat message from a remote GSM. Using this information, the IO Proxy tries to guess a potential node failure before the GSM makes a more concrete decision. If the IO Proxy decides that a node has potentially failed and the decision turns out to be correct, then the node will ultimately be removed by the GSM from the system and appropriate action will be taken. In the meantime, the distributed group appears to respond quickly to the failed node by predicting the failed node in advance. If, however, the IO Proxy was incorrect and the node was only unavailable during a period of time due to network congestion or a high load on the node, all connections to the node would have been terminated. The result of such an action could have lead to an aborted transaction, or the termination of any client query connections. Once the temporary communication problem with the node resolves itself, all GSM controllers within the distributed group will resume receiving heartbeat messages from the node. At this point, the node will no longer be classified as potentially failed and inter-node communication involving the node will be permitted.

Finally, there are some subtle problems caused by using the IO Proxy. The first such problem is that since all virtual incoming remote connections come directly from the local node controller, it is not possible to determine the remote host of the endpoint by simply looking at the other endpoint of the connection. This has implications for the node controller which will be discussed in the implementation section. Secondly, if an IO error occurs with the remote component of a virtual connection, it is possible that the local component will not be made aware of the error. This is especially true when writing data over the connection. The reason is for this is because the direct connection to the local IO Proxy may be problem free, while some other part of the virtual connection is not. If it is important to have confirmation that data was received by the remote component, the local component should first send the data and then wait for a response.

3.3.3 Four11 Server

Resource location is always an issue in any distributed system. In some cases, the resources in the distributed system are known *a priori*, while in other cases, resources are dynamic and volatile. When all resources are known *a priori*, a simple solution is to manually specify the resources required for a particular service. This can be done by supplying a configuration file describing all resources any component might need. We use the term resource loosely, but in the context of the RDSS, a resource is typically the TCP port number where a particular service can be located. Limited to the scope of locating port numbers, the Four11 server performs a similar task as the RPC portmap service [16].

The feasibility of manually specifying resource locations depends on a number of factors. We have identified three necessary conditions for manual specification of resources to be successful. The first requirement is that it must be physically possible to manually assign resources in the system. Secondly, there must be a limited number of resources to specify and finally, it is important for the distributed system to remain relatively stable. The violation of any of these three requirements makes manual configuration difficult, if not infeasible. MultiText, the distributed IR system used as a model for our research, uses a configuration file to solve the resource location problem within the query marshaler.

While this solution has worked for the MultiText system, it has been successful only because the number of application instances in the MultiText system is extremely stable and so reconfiguration is seldom needed. The RDSS design violates two, if not all of the feasibility requirements of manual resource specification. Firstly, it is not possible to manually assign TCP port numbers to application instances with 100 percent certainty. The RDSS does not assume nor require that any node in the distributed group membership is not being used for any other purposes other than those required by the RDSS. As a result, it is not always possible to guarantee that any manually assigned port numbers have not already been used by another process. It should be noted that manually assigning port numbers in the MultiText system suffers this same problem. The problem is less of an issue with MultiText because the number of ports to specify is much smaller and so the probably of a conflict occurring is smaller.

Secondly, the number of ports to manually assign grows quadratically with the number of nodes in the system. In our prototype implementation running with 12 nodes, each node would require approximately 30 port numbers to be manually assigned and configured. Obviously, this solution does not scale beyond a handful of nodes participating in the distributed group. It is not clear that an RDSS distributed system is less stable than a non-RDSS distributed system. We have taken the pessimistic view that an RDSS distributed system is not stable in the design of the RDSS.

For the reasons listed above, the RDSS uses an automated resource location service.

The Four11 server allows clients to register, release and query the port numbers of specific components in the RDSS. No persistent state is maintained in the Four11 server since the port numbers would be meaningless after a node failure. There is one Four11 server per node in the system. Each Four11 server is responsible for maintaining information about the components running on the local node only. Communication with the Four11 server is initiated using a well known, TCP port.

Three major services are provided by the Four11 server. These operations are *register*, *release* and *query*. Before a client can initiate contact with another RDSS component, the component to be contacted must first register the TCP port on which it is listening for incoming connections with its local Four11 server. Once registered, a client will contact the local Four11 server of the component to be contacted and query the TCP port number of the service to be contacted. After learning the TCP port number from the Four11 server, the client can now contact the component offering the service directly. When the component is done using the port, it releases the port by informing its local Four11 server. It should be noted that simply registering a port does not guarantee the ability to communicate using that port. The Four11 server is much like a phone directory. If the wrong phone number is published, you will not be able to call the person you looked up.

Going back to the example of the query marshaler. In the RDSS, the query marshaler, when contacting the necessary application instances, must now contact the Four11 server on each node to find the TCP port number that should be used when contacting the primary application instance on each node. In the event of a node failure, this process is complicated by also having to query for both the local primary application instance as well as the foreign secondary application instance of the failed node. Figure 3.5 shows an example of the a query marshaler establishing connections with all primary application instances in a system with 4 nodes. The numbers in the figure correspond to the relative temporal ordering of the events. The first event to occur is a client connecting to the query marshaler. The second event is the query marshaler requesting the TCP port numbers of each application instance from its local Four11 server. In the third event, the query marshaler waits for all Four11 servers to respond. Finally, the fourth event is the establishment of connections to each of the application instances.

3.3.4 GSM Controller

The primary purpose of the GSM Controller is to run the GSM algorithm described in Chapter 2. There is one GSM Controller running on each node in the system. To run the GSM algorithm, the GSM Controller must be able to transmit the GSM's view of the distributed group to all GSMs in the system and likewise must be able to receive information from each remote GSM. Information is transmitted between GSMs in the

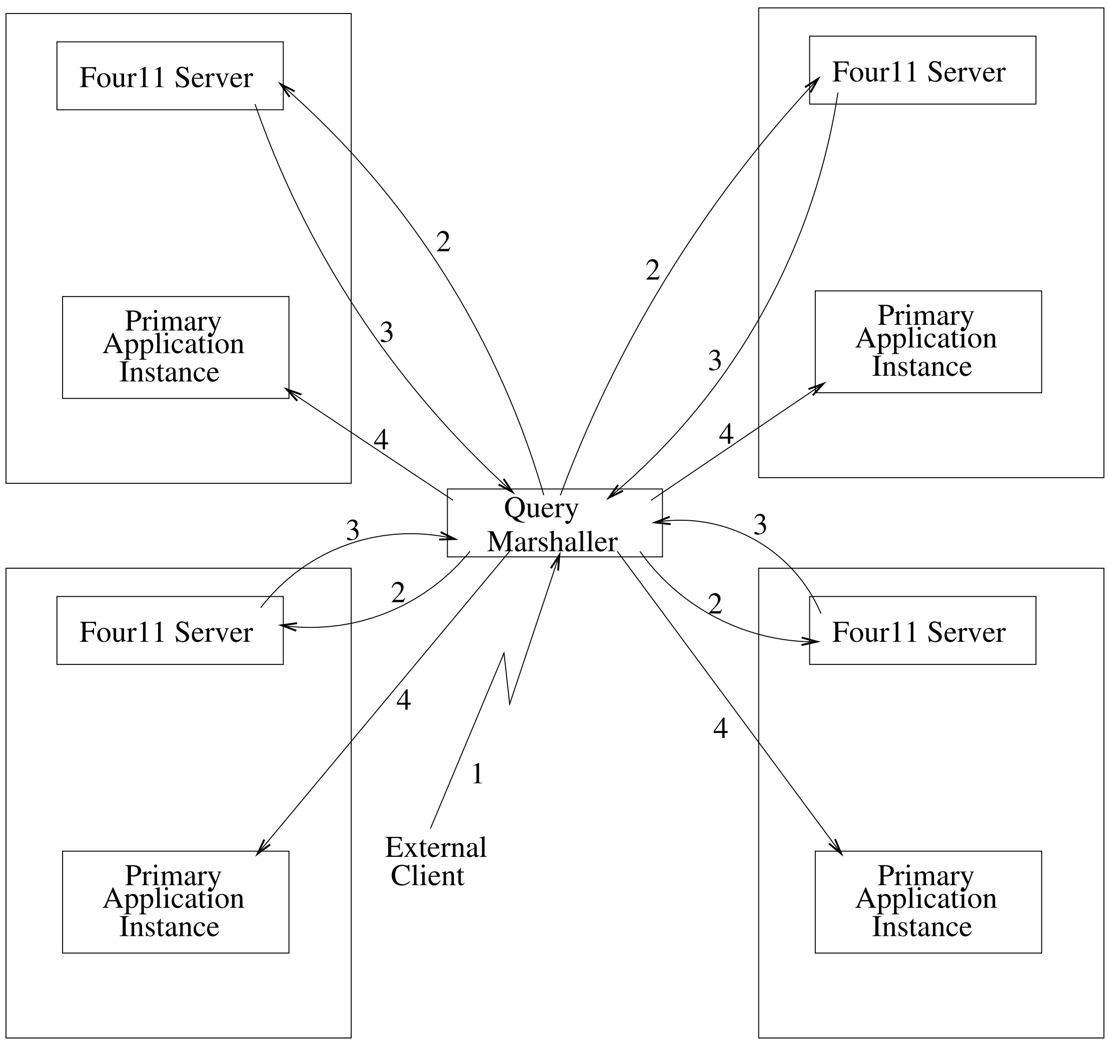

Figure 3.5: Four11 Server Interaction

form of a heartbeat message. Each heartbeat message contains the source GSM's view of the current global state, the number of attempted transactions, the number of committed transactions and the amount of time since the GSM has last heard from each GSM in the distributed group membership. The source GSM is the GSM that create the heartbeat.

Transmission of GSM state information is performed by broadcasting the heartbeat message in a UDP packet. TCP could have been used, but two reasons make UDP a better choice. First, by broadcasting the heart beat message using UDP, the heartbeat message can be sent to all GSMs in the system with a single transmission. Transmitting the heartbeat message using TCP would require retransmitting the same heart beat message to each individual GSM. The second reason that makes TCP unsuitable is the time-sensitive nature of heartbeat messages. The guarantee of delivery provided by TCP is not needed, nor is the associated overhead. If, for some reason, a heartbeat message can not be delivered immediately to a GSM, a new heartbeat message will be generated and sent momentarily. Once the new heartbeat message is generated, we do not care if the old heartbeat message ever arrives because it is no longer a valid description of the GSM's view of the distributed system. Furthermore, should TCP be used, it is theoretically possible that in the process of transmitting a heartbeat message to each remote GSM, a new heartbeat will be generated and the heartbeat message currently being transmitted is no longer a valid description of the local GSM's view of the distributed group. A final reason for using UDP pertains to resource discovery. If a new node wishes to be added to the distributed group membership, the node must have some mechanism to initiate communication with the existing group. Without knowledge of this group, the new node will be unable to advertise itself to the group. The combination of UDP and a broadcast address provides a trivial solution to this problem.

In addition to exchanging heartbeat messages between GSMs, there is a need for other components in the RDSS to inquire about the current state of the GSM. As the GSM controller is the only RDSS component with such knowledge, the GSM controller must make available its view of the distributed system to any component requiring this information. Any component in the RDSS can connect to the GSM controller using a well known TCP port and request a copy of the next heart beat message generated for transmission.

Finally, looking back at figure 2.1, we see that after reaching a $PREPARE$ state, the next transition is to either a $COMMIT$ or an $ABORT$ state. When the GSM enters a $PREPARE$ state, the RDSS should proceed to perform any work required for the current operation. The GSM algorithm will not transition out of the $PREPARE$ state without external input regarding the success or failure of the work performed as a result of the $PREPARE$ state transition. To allow for this external input, the GSM controller accepts an additional connection to allow one external component to provide this information. In the RDSS, the external agent providing this information is the Node Controller. We will visit the Node Controller later in this chapter.

The interaction between the GSM controller and the GSM algorithm is limited to a well defined API. The GSM controller must pass any arriving heartbeat messages to the GSM and must be able to generate new heartbeat messages periodically. The API used by the GSM controller to interact with the GSM algorithm is defined in chapter 5.

3.3.5 Distribution Server

Recall that one of the goals of the RDSS was to incorporate redundancy in the distribution of documents in the IR system to allow the system to suffer no permanent loss of data, nor any temporary loss of data availability, in the event of a single node failure. The redundancy incorporated in the distributed system is explained later in this chapter and is called the *redundant data distribution scheme* (RDDS). It is the job of the Distribution server to ensure that the distribution of the document collection in the system adheres to the redundant data distribution scheme. A secondary task of the distribution server is to maintain a balanced distribution of the documents in the distribution system. A balanced distribution is essential for performance since this slowest node is the bottleneck when executing client queries.

There is one distribution server on each node in the distributed group. The distribution servers are designed to function in parallel with each other. When performing any group operation, no single distribution server is responsible for all work to be performed. This design provides better performance since all data movement in the system can be computed by n processes, and not just a single process.

Recall that the primary application instance is responsible for storing and indexing a subset of the global document collection, while the secondary application instances are used for replication only. The local distribution server is responsible for ensuring the validity of the RDDS involving only the documents being indexed by the primary application instance of the local node. This means that each distribution server must manage the distribution of documents between the primary application instance and all remote secondary application instances (replication). There are two reasons for isolating the RDDS of each primary document collection from each other. The first is simplicity. It is much simpler to manage the distribution of documents across one primary application instance and $n-1$ remote secondary application instances then it is to manage n primary application instances and $n(n-1)$ secondary application instances. The second reason for isolating the RDDS of each primary document collection from each other is performance. Parallelizing the work to be done by the distribution servers allows the performance of the system to scale with the size of the system.

It is important to note that, collectively, all distribution servers in the distributed group are required to guarantee a global document distribution that adheres to the RDDS. Failure to do so could lead to a temporary loss of data availability or even a permanent loss of data in the event of a node failure or removal. Referring back to the four RDDS axioms, axiom 1 states that the set of documents stored in each primary application instance are pairwise disjoint. The consequences of this axiom are that the set of documents administered by any two distribution servers are disjoint and hence the actions of one distribution server will not interfere with those of another. Achieving a global document distribution that adheres to the RDDS is accomplished by instructing each distribution server to ensure a valid RDDS of the document set for which it is responsble (axioms 3 and 4). The only exception is when new documents are added to the global document collection. In this case, one distribution server is given the responsibility of guaranteeing a valid RDDS of the global document collection across all primary application instances in addition to its normal responsibilities. In figure 3.1, the distribution server on node 1 would be responsible for all documents in application instances $P1$, $S21$, $S31$, and $S41$.

Although desirable, we do not require the distribution server to guarantee a balanced distribution of the global document collection. A best effort is made to ensure the global document collection is evenly distributied, however, since no single distribution server decides the location of every document in the global document collection, an unbalanced distribution of documents can occur under certain circumstances. When adding and removing nodes, the distribution servers are able to guarantee a balanced distribution of the global document collection. Document collection operations are what can cause an even distribution to become uneven. A large number of document deletions from one, or only a few nodes, could lead to one or several nodes indexing a disproportionate number of documents when compared to other nodes in the system. In this case, the distribution servers will still guarantee a valid RDDS distribution; however, performance may suffer as a result of the unbalanced distribution.

3.3.6 Node Controller

We have already seen that the GSM Controller is responsible for running the GSM algorithm and making the GSM state available to be queried by other components. Other than running the algorithm, the GSM controller does nothing productive once the GSM makes a decision. When the GSM state enters one of the $PREPARE$ states, it will not transition to a $COMMIT$ or an $ABORT$ state unless it is told to do so or if a node fails. This is where the Node Controller comes in. The Node Controller continually monitors the GSM Controller, and in particular, pays attention to the current state of the GSM. As soon as the GSM transitions into a $PREPARE$ state, the Node Controller proceeds to guide the node through all necessary actions to accomplish the action dictated by the GSM. Once all work associated with the specified operation has been completed, the node controller informs the GSM controller of its ability to commit or abort the operation. This information is then passed on to the GSM algorithm by the GSM controller.

There is one node controller on each node in the distributed group. The node controller is much like a conductor of an orchestra. The conductor does not play the instruments, but rather gives indications to various orchestra members when it is their turn to do some work. The node controller plays a similar role. In the this section, we introduce some of the requirements of the node controller in order to properly "conduct" a node addition, node removal or document collection update.

The tasks involved in performing an operation as dictated by the GSM algorithm are many, with some being very complex. Many race conditions exist that must be handled carefully for the node controller to operate successfully. Coordinating a distributed algorithm is non-trivial and so the node controller acts as a centralized organizational component whose sole task is to guide all components in the system in a constructive manner. Here we describe, in general, the various actions performed by the Node Controller. In Chapter 5, we describe these actions in more detail and specify the exact order.

All operations are initiated by the GSM algorithm making a transition into one of the $PREAPRE$ state. When a node fails, or a new node is detected, the necessary transitions take place automatically. Document collection updates can not occur without external input requesting an update operation to occur. The steps required for an external client to request an update operation to occur begin with the client contacting the node controller and requesting an update operation. The client must block until given permission by the node controller to proceed. The node controller then uses initiates an update operation via the GSM controller. Eventually, the GSM of the local GSM controller will transition into the $UPDATE\_PREPARE$ state and the local node controller will see this transition, as will all node controllers in the system. At this point, the update operation is in progress and the client will ultimately be told that it may proceed with the update.

Node Controller Synchronization

The node controller is responsible for guiding the node through all necessary actions that must be performed for a given operation. At times, race conditions exist that must be avoided in order to ensure a feasible execution path for performing the necessary actions. An example of such a race condition is the spawning of new application instances. When an application instance is spawned, it must set up TCP listening ports to accept incoming connections. It is imperative that no other component in the system attempt to contact the newly spawned application instances until the TCP listening ports have been properly established.

Node controller synchronization should not be confused with GSM synchronization. They are both similar in that both require a distributed concensus. The difference is in terms of the service offered. GSM synchronization offers the service of an agreed upon state of the distributed system (group membership, current operation). The distributed concensus takes place between all GSM controllers within the system. Node Controller synchronization provides a service whereby each of the node controllers in the system can indicated that a minimum amount of work has been performed by each node controller. Node controller synchronization involves communication between node controllers only. Furthermore, node controller synchronization uses some of the services provided by the GSM.

Node controller synchronization is quite simple. For each distributed transaction to be performed, one node controller is appointed the role of *master* node controller while all others become *slave* node controllers. When a synchronization point has been reached, all slave node controllers will contact the master node controller indicating that it has reached the synchronization point. No node controller is allowed to proceed until all node controllers have indicated that they too have reached the synchronization point. Once the master node controller has heard from all slave node controllers, and has itself, reached the synchronization point, the master node controller will respond to each slave node controller, explicitly giving permission for each node controller to proceed beyond the current synchronization point.

Some discussion is warranted to describe the case when a node failure occurs during synchronization. Recall that all inter-node communication involves the use of the IO Proxy. If a node failure occurs, all IO Proxies in the system will close any connections involving components on the failed node, including the node controller. If a slave node controller fails, the master node controller will notice the prematurely closed connection and deduce that some form of error has occurred. In this situation, the master node controller will indicate to all remaining slave node controllers that an error has occurred and the current operation is to be aborted. The situation is much the same if the master node controller fails. When the master node controller fails, all connections from the slave node controllers to the master node controller will be prematurely closed. Much like the former case, each of the slave node controllers will know that an error has occurred and will proceed to abort the current operation.

An extension to the preceding protocol already alluded to is to allow a vote to take place at each synchronization point. Each node controller is voting whether or not it wants the transaction to proceed. The voting procedure that takes place is similar to the voting procedure that takes place during a two phase commit. Each node controller can vote to either proceed with, or abort, the current transaction. The transaction can proceed if and only if all node controllers vote to proceed with the transaction. It should be noted that such a vote has the potential of avoiding fruitless work, but is not necessary. If a single node can not proceed with a transaction but the remaining nodes can, we could just have the synchronization point take place without a vote. Eventually, the GSM Controller will be told by the one node that the distributed transaction should be aborted and as a result, the GSM will decided to abort the distributed transaction. All other node controllers will have to abort any work they performed as part of the transaction. By holding a vote at each synchronization point, we give the node controllers a chance to realize sooner, rather than later, that the distributed transaction will be aborted, allowing each node controller to avoid spending time on an operation that is destined to fail.

Node Controller Operations

Regardless of the operation being performed, there are several common activities that must take place. A connection to the local Four11 server is needed in order to be able to locate any of the application instances running on the local node. The distribution server must also be contacted because once the node controller has prepared all other components to accept work, the distribution is then allowed to perform all of its tasks. The distribution server is where most of the work is done when performing node operations, but it is only a small piece of the puzzle. Finally, one connection is needed to the king node controller for synchronization purposes.

In addition to instructing other RDSS components to do work, the node controller is responsible for progressing the application instances through the various stages of the transaction. Although it is acceptable to do so, the application does not need to be able to guide itself through a transaction. The node guides the application instances through every step of a transaction. As a result, the node controller must store some minimal state when performing a transaction to allow for error recovery

3.4 Algorithms

In this section, we describe three system operations and the necessary tasks that must take place. The first two operations deal with the distributed group membership. Adding or removing a node from the distributed system involves a change in system resources and a corresponding need to redistribute the global document collection according to the RDDS. Document collection updates modify the global document collection by removing old documents and adding new documents. Regardless of what operation is being performed, it is required that a valid RDDS distribution exist at the completion of the operation. It is desirable that a balanced valid distribution exist, though, there is no requirement that the distribution is balanced. All algorithms are described from the perspective of a single node. We always assume that at the beginning of an operation, a valid RDDS distribution is in place. We make no assumptions regarding whether or not a distribution is balanced. At the termination of the operation, we must minimally guarantee that a new, valid RDDS distribution exists.

The distribution server is solely responsible for following the algorithms described in this section. The description of the node removal and node addition operations do not discuss the operation from the perspective of the distribution server. When describing the document collection updates, we will make reference to the distribution servers on a particular nodes.

3.4.1 Node Removal

Let the current distributed membership contain n nodes and the current operation is to remove one of those nodes. We will refer to the node being removed from the distributed membership as the *failed* node because this is the decision that must have been made in order for the GSM to choose to remove the node in the first place Let the failed node be denoted by N_f and the current node be denoted N_c. We outline the impact of the failed node on the distributed of the global document collection and the violation of any of the four axioms of a valid RDDS distribution. We then describe the steps taken by a single node, with all nodes performing the same steps in parallel. Finally, we prove that, through the collective efforts of all nodes, a valid RDDS distribution is in place at the completion of the operation. Figure 3.6 shows a RDSS system with five nodes. We use this figure as a base figure to help illustrate the data movements involved when removing one of the five nodes.

Distribution Impact

There are two problems that must be solved in removing a node. The first is that because a node has failed, axiom 3 of the RDDS definition is violated for each of the remaining $n-1$ primary application instances. The second problem is that axiom 1 of the RDDS definition is also violated. Table 3.3 shows the the current state of the global document collection in terms of the violated axioms 1 and 3. Finally, we would like to redistribute the documents as evenly as possible to maximize performance of the distributed system.

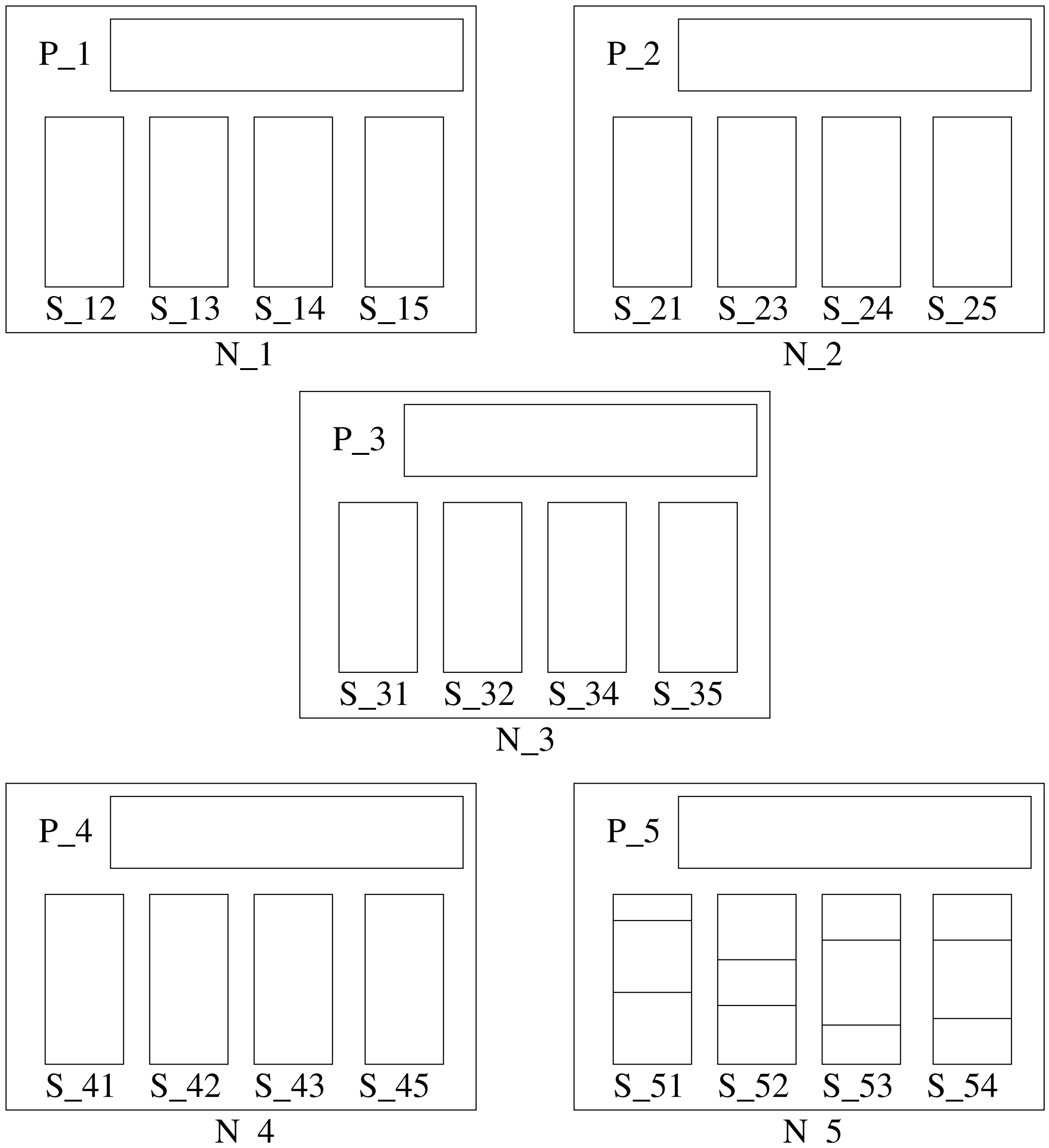

Figure 3.6: Base RDSS system with 5 nodes

$$\bigcup_{i=1, i \neq f}^{n} D(P_i) = D_g - D(P_f)$$

$$\forall i, i \neq f, \quad \bigcup_{j=1, j \neq f, j \neq c}^{n} D(S_{ji}) = D(P_i) - D(S_{fi})$$

Table 3.3: Node Removal RDDS Axiom Violation

Steps

The first step in removing a node from the distributed group membership is to re-establish adherence of axiom 3 of the RDDS definition. Looking at table 3.3 and setting $i = c$, to re-establish axiom 3, we need to move the set of documents stored in S_{fc} from the right hand side of the equation to the left hand side of the equation. There are $n-2$ remaining remote application instances in the system. The solution is to partition $D(S_{fc})$ into $n-2$ subsets. Let the subsets be denoted $s_i, 1 \leq i \leq n, i \neq f, i \neq c$. Each subset is added to the document collection of one of the remaining remote application instances. The subset s_i is added to the remote application instance S_{ic}. Figure 3.7 shows how axiom 3 is restored during a node removal operation.

The second step in removing a node from the distributed membership is to re-establish adherence to axiom 1 of the RDDS definition. Again, referring back to table 3.3, there is a need to move all documents stored in P_f from the right hand side of the equation to the left hand side. The solution is to partition $D(P_f)$ into $n-1$ subsets. Let the subsets be denoted $p_i, 1 \leq i \leq n, i \neq f$. The node now chooses one of these subsets to add to the document collection of P_c. The subset chosen by the node is p_c. Figure 3.8 shows how axiom 1 is restored during a node removal operation.

At this point, we claim to have fixed both axioms 1 and 3. In fact, while we did correct axiom 3 during step 1, we broke it again during step 2. Axiom 3 is now violated because the subset p_c was added to the primary application instance P_c but it was not replicated in any of the remote application instances. The third step in removing a node is to redistributed the subset p_c across all remaining remote application instances. We partition p_c into $n-2$ subsets. Let the subsets be denoted $p_{ic}, 1 \leq i \leq n, i \neq f, i \neq c$. Each subset is is added to the document collection of one of the remaining remote application instances. The subset p_{ic} is added to the remote application instance S_{ic}. Figure 3.9 shows how axiom 3 is re-restored during a node removal operation.

The final step to removing a node from the distributed membership is that after all documents have been redistributed and the operation was successful (according to the GSM), the foreign secondary application instance associated with the failed node must be destroyed and any resources reclaimed. In chapter 5, we examine how the document collection redistribution can be performed efficiently.

Global Result

Now that the remaining $n-1$ nodes have performed their individual tasks, we now look at the collective accomplishments of all $n-1$ nodes. We will prove that all axioms are now satisfied to prove that a valid RDDS exists at the completion of the operation. During

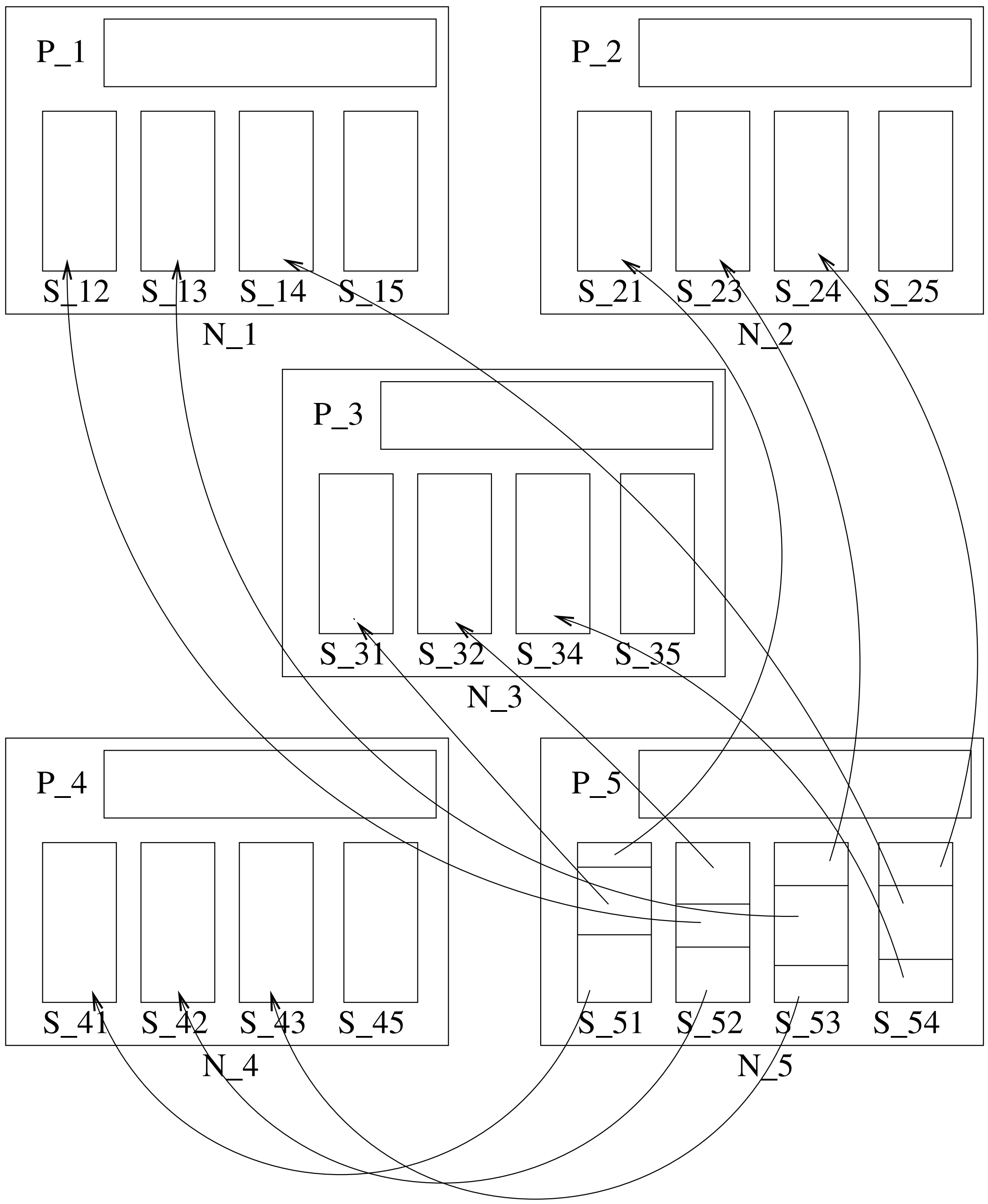

Figure 3.7: Node Removal: Step 1

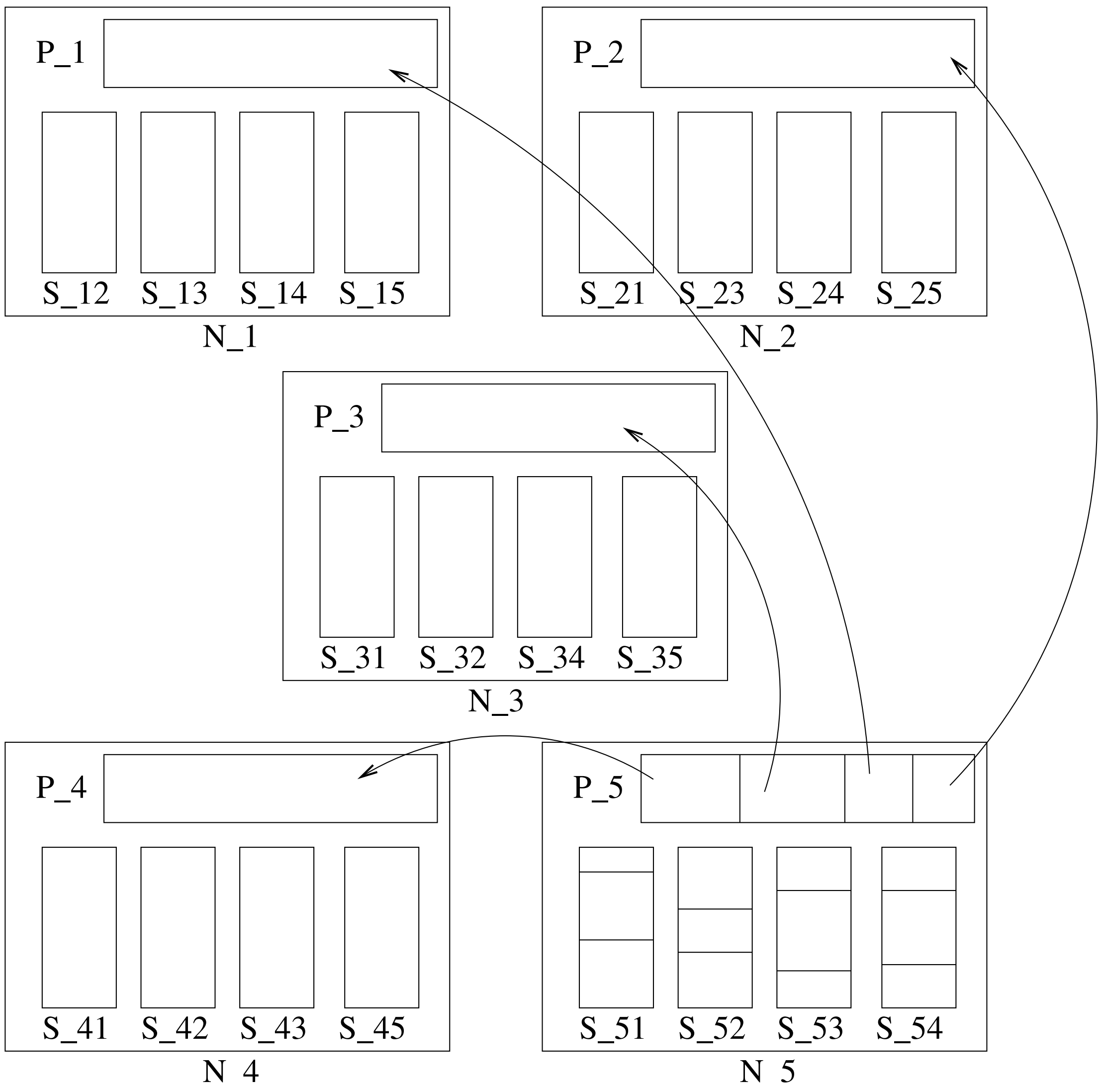

Figure 3.8: Node Removal: Step 2

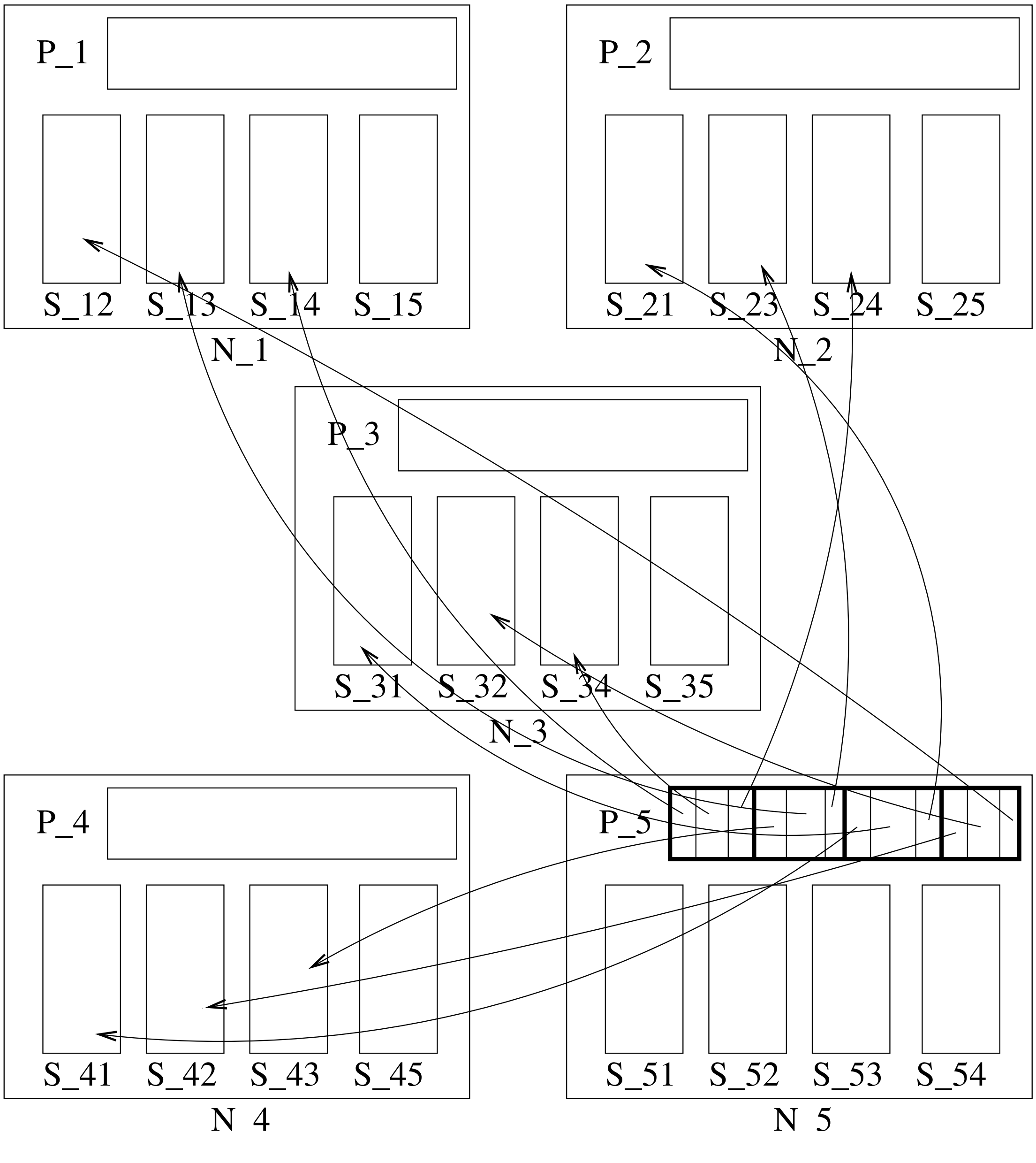

Figure 3.9: Node Removal: Step 3

the operation, the document collections stored by the primary and secondary application instances were modified. We use the notation $D(A)$ to represent the document collection stored by application instance A before the operation began. $D'(A)$ is the set of documents stored by the application instance A after the operation is finished. Table 3.4 summarizes all changes made to the application instance document collections. We now prove the four axioms of the RDDS definition are true at the completion of a node removal operation.

$$D'(P_c) = D(P_c) \bigcup p_c$$
$$D'(S_{ic}) = D(S_{ic}) \bigcup s_i \bigcup p_{ic}$$

Table 3.4: Node Removal Document Collection Changes

Axiom 1 states that the global document collection must be distributed across all primary application instances. The proof showing axiom 1 is true at the completion of the operation is shown in table 3.5.

$$\begin{aligned}
\bigcup_{i=1}^{n-1} D'(P_i) &= \bigcup_{i=1, i\neq f}^{n} (D(P_i) \bigcup p_i) \\
&= \bigcup_{i=1, i\neq f}^{n} D(P_i) \bigcup (\bigcup_{i=1, i\neq f}^{n} p_i) \\
&= \bigcup_{i=1, i\neq f}^{n} D(P_i) \bigcup D(P_f) \\
&= D_g
\end{aligned}$$

Table 3.5: Node Removal Axiom 1 Proof

Axiom 2 states that the document collections of each of the primary application instances must be pairwise distinct. To prove this axiom, we choose two arbitrary primary application instances and show that their document collections are disjoint. The proof is shown in table 3.6.

Axiom 3 states that the document collection stored by each primary application instances must be distributed over all remote application instances. We choose an arbitrary primary application instance and show that axiom 3 is true for that instance. Since we choose an arbitrary application instance, it is true for all primary application instances. Let P_i be the arbitrary application instance. The proof is shown in table 3.7.

Axiom 4 states that for all nodes in the system, the document collections stored by the remote application instances must be pairwise distinct. We choose an arbitrary node and

$$
\begin{aligned}
D'(P_i)\bigcap D'(P_j) &= \{D(P_i)\bigcup p_i\}\bigcap\{D(P_j)\bigcup p_j\} \\
&= \{D(P_i)\bigcap D(P_j)\}\bigcup\{D(P_i)\bigcap p_j\}\bigcup\{p_i\bigcap D(P_j)\}\bigcup\{p_i\bigcap p_j\} \\
&= \emptyset\bigcup\emptyset\bigcup\emptyset\bigcup\emptyset \\
&= \emptyset
\end{aligned}
$$

Table 3.6: Node Removal Axiom 2 Proof

$$
\begin{aligned}
\bigcup_{j=1}^{n-1} D'(S_{ji}) &= \bigcup_{j=1,j\neq f,j\neq i}^{n} (D(S_{ji})\bigcup s_j\bigcup p_{ji}) \\
&= \bigcup_{j=1,j\neq f,j\neq i}^{n} D(S_{ji}) \bigcup_{j=1,j\neq f,j\neq i}^{n} s_{ji} \bigcup_{j=1,j\neq f,j\neq i}^{n} p_{ji} \\
&= (D(P_i) - D(S_{fi}))\bigcup D(S_{fi})\bigcup p_i \\
&= D(P_i)\bigcup p_i \\
&= D'(P_i)
\end{aligned}
$$

Table 3.7: Node Removal Axiom 3 Proof

show that axiom 4 is true for that node. Proving the axiom holds for an arbitrary node proves the axiom holds for all nodes. Let i be the arbitrarily chosen node. The proof is shown in table 3.8.

Subset Selection

So far, our description of the various subsets has not enforced any restrictions regarding the size or makeup of the subsets. It is permissible for some of the the subsets chosen to be empty. While correctness would be maintained, one of our goals is to maintain a balanced distributione where possible. In light of this goal, our subset selection heuristics are designed to evenly partition any document collection into balanced partitions where each partitions contains approximately the same number of documents. We should note that documents may not be the same size. However, if the size of the largest document is small compared to the storage capacities of the nodes, the differences in storage requirements between the various partitions should be negligeable.

It turns out that selecting balanced subsets is a trivial task during a node removal operation. The main reason for this is that all document transfers that take place involved adding documents and not removing documents. The removal of documents greatly complicates the process of maintaining balanced distributions since it is necessary to know

$$
\begin{aligned}
D'(S_{ji}) \bigcap D'(S_{ki}) &= \{D(S_{ji}) \bigcup s_j \bigcup p_{ji}\} \bigcap \{D(S_{ki}) \bigcup s_k \bigcup p_{ki}\} \\
&= \{D(S_{ji}) \bigcap D(S_{ki})\} \bigcup \{D(S_{ji}) \bigcap s_k\} \bigcup \{D(S_{ji}) \bigcup p_{ki}\} \\
&\cup \ \{s_j \bigcap D(S_{ki})\} \bigcup \{s_j \bigcap s_k\} \bigcup \{s_j \bigcup p_{ki}\} \\
&\cup \ \{p_{ji} \bigcap D(S_{ki})\} \bigcup \{p_{ji} \bigcap s_k\} \bigcup \{p_{ji} \bigcap p_{ki}\} \\
&= \emptyset \bigcup \emptyset \bigcup \emptyset \\
&\cup \ \emptyset \bigcup \emptyset \bigcup \emptyset \\
&\cup \ \emptyset \bigcup \emptyset \bigcup \emptyset \\
&= \emptyset
\end{aligned}
$$

Table 3.8: Node Removal Axiom 4 Proof

where a particular document is located to avoid removing a disproportionate number of documents from a single component.

Step one of the node removal operation involves each node partitioning the document set $D(S_{fc})$ into $n-2$ partitions. Since we are looking for evenly sized partitions, each partition should be approximately $1/n-2$ of the document collection $D(S_{fc})$. Each of these partitions is then added to exactly one of the remaining remote secondary application instances. Furthermore, each secondary remote application instance receives exactly one partition.

The second step in removing a node is to partition the document set P_f into $n-1$ partitions. Again, since we are looking for balanced partitions, each partitions should be approximately $1/n-1$ of the document collection $D(P_f)$. Each of these partitions are added to exactly one of the remaining primary application instances. Just as before, each primary application instances accepts exactly one of these partitions.

The final step is to further partition the subset P_c into $n-2$ evenly sized subsets. The subset P_c is the subset of $D(P_f)$ that was selected in step two and was added to the local primary application instance. After creating the $n-2$ partitions, each partition is added to exactly one of the remaining $n-2$ remote secondary application instances just like step one.

We now give a brief explanation showing that assume we begin with a balanced valid RDDS distribution, our subset selection heuristics will maintain a balanced distribution. Let us assume that our global document collection consists of d documents. Table 3.9 shows the estimated document collection sizes for primary and secondary application instances in a RDSS system wtih n nodes.

Referring back to our three steps, in step one, the document collection $D(S_{fc})$ contains

Application Type	How Many	Collection Size
Primary	n	d/n
Secondary	$n(n-1)$	$d/n(n-1)$

Table 3.9: Component Document Collection Sizes

approximates $d/n(n-1)$ documents. Partitioning this colleciton in $n-2$ balanced partitions leaves each paritition hold approximately $d/n(n-1)(n-2)$ documents. In Step two, we beging with $D(P_f)$ which contains approximately d/n documents. Each partition created in step two will therefore contain approximately $d/n(n-1)$ documents. Finally in step three, each partition created in step two is further divided into $n-2$ smaller partitions. Each of these partitions will be approximately $d/n(n-1)(n-2)$ documents in size.

After creating all partitions and performing all data movement operations described in the three steps, we now show the new estimated sizes of the document collections for both primary and secondary application instances. We assume a balanced distribution exists before any data movement takes place. The proofs demonstrate that our subset selection maintains a balanced distribution. Table 3.10 shows the estimated size of the primary application instances after the operations and table 3.11 shows the same for the secondary application instances. We note that these sizes correspond to the values shown in table 3.9 for a RDSS system with $n-1$ nodes.

$$\begin{aligned} |D'(P_i)| &= |D(P_i)| + |p_i| \\ &= d/n + (d/n(n-1)) \\ &= d/n-1 \end{aligned}$$

Table 3.10: Primary Application Size

$$\begin{aligned} |D'(S_{ji})| &= |D(S_{ji}) + |s_i| + |p_{ic}| \\ &= d/n(n-1) + d/n(n-1)(n-2) + d/n(n-1)(n-2) \\ &= d/(n-1)(n-2) \end{aligned}$$

Table 3.11: Secondary Application Size

3.4.2 Node Addition

In this section, we describe the actions that must take place when adding a node to the distributed membership. Let N_a denote the node being added and N_c be the current node. Similar to the node removal operation, we first outline the distribution impacts, describe the steps taken by each node and the finally prove that a valid RDDS is in place at the end of the operation. Since all work performed when adding a new node is motivated by the need to evenly distribute the document collection, after proving our document relocations do not violate the RDDS definition, we will give a brief analysis showing why the document relocations that take place ultimately lead to a balanced distribution.

Distribution Impact

Adding a new node does not violate any of the axioms of the RDDS definition. If all we cared about was data availability and replication, it would be perfectly acceptable to do nothing when a new node is added to the system. The RDDS would still guarantee no oss of data in the event of a single node failure. However, automatic load balancing is also of the design criteria for the RDSS and so a redistribution of the document collection is performed whenever a new node is added to the system.

The two main problems to solve when adding a new node is that the distribution of the global document collection over all primary application instances is no longer balanced. Currently, the global document collection is distributed over n primary application instances. The new node offers an additional primary application instance that can be used to reduce the load of each of the previously existing n primary application instances. The second problem to solve is that the document collection of each of the previously existing primary application instances is distributed across $n-1$ remote application instances. The new node offers an addition remote application instance and again, it would be beneficial to make use of the new node's resources.

Steps

The first step in adding a node to the distributed group membership is to redistribute the primary document collection over all remote secondary application instances. The addition of a new node to the group membership provides an additional remote secondary application instance. A subset of the documents of the primary application instance must be chosen to be relocated to the new remote application instance. Additionally, this same subset must be removed from the previously existing $n-1$ remote application instances or axiom 4 of the RDDS definition will be violated. Let r_c be the subset of documents chosen from P_c that that will be replicated in S_{ca}. Let $r_{ic}, 1 \leq i \leq n+1, i \neq c, i \neq a$ be the subset of r_c that is currently stored in the remote application instance S_{ic}. It should be noted that $r_{ic}, 1 \leq i \leq n+1, i \neq c, i \neq a$ form a partition of r_c (axiom 3 was true at the beginning of the operation). r_{ic} will be removed from the document collection of S_{ic}.

The second step to be performed is to redistribute the global document collection across all primary application instances. A subset of the primary application instance is chosen to be removed from P_c and added to P_a. Let p_c be this subset. We require that $p_c \bigcap r_c = \emptyset$. When removing the subset p_c from P_c, we must also remove all documents in p_c from the remote application instances. Let $p_{ic}, 1 \leq i \leq n+1, i \neq c, i \neq a$ be the the subset of p_c that is currently stored in the remote application instance S_{ic}. The subsets $p_{ic}, 1 \leq i \leq n+1, i \neq c, i \neq a$ form a partition of p_c (axiom 3). Similarly, When adding the subset p_c to P_a, we must also distributed all documents in p_c to the remote application instances of node N_a. To simplify the distribution across all remote application instances, the entire set of documents p_c are all added to the foreign application instance S_{ia}.

Global Result

Just like the analysis of a node removal operation, we now look at the the state of the distribution of the global document collection across all $n+1$ nodes that now exist. We use the same notation already introduced. $D(A)$ and$D'(A)$ are used to represent the document collections stored by application instance A before and after the completion of the operation, respectively. Table 3.12 shows all changes made to the document collections of the different application instances in the system.

$$\begin{aligned}
D'(P_i) &= D(P_i) - p_i, 1 \leq i \leq n+1, i \neq a \\
D'(S_{ji}) &= D(S_{ji}) - r_{ji} - p_{ji}, 1 \leq i, j, \leq n+1, i \neq a, j \neq i, j \neq a \\
D'(S_{ai}) &= r_i, 1 \leq i \leq n+1, i \neq a \\
D'(P_a) &= \bigcup_{i=1, i \neq a}^{n+1} p_i \\
D'(S_{ia}) &= p_i, 1 \leq i \leq n+1, i \neq a
\end{aligned}$$

Table 3.12: Node Addition Document Collection Changes

Axiom 1 requires us to show that the global document collection is distributed across all primary application instances. The proof is shown in table 3.13.

The proof of axiom 2 is done in two parts. First, we consider any two primary application instances that existed before the node addition operation was performed. Secondly, we consider any one of the previously existing primary application instances with the new

$$
\begin{aligned}
\bigcup_{i=1}^{n+1} D'(P_i) &= (\bigcup_{i=1, i \neq a}^{n+1} D'(P_i)) \bigcup D'(P_a) \\
&= (\bigcup_{i=1}^{n} D(P_i) - p_i) \bigcup_{i=1}^{n} p_i \\
&= \bigcup_{i=1}^{n} D(P_i) \\
&= D_g
\end{aligned}
$$

Table 3.13: Node Addition Axiom 1 Proof

$$
\begin{aligned}
D'(P_i) \bigcap D'(P_j) &= \{D(P_i) - p_i\} \bigcap \{D(P_j) - p_j\} \\
&= \{D(P_i) \bigcap D(P_j)\} \\
&= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
D'(P_i) \bigcap D'(P_a) &= \{D(P_i) - p_i\} \bigcap \{ \bigcup_{j=1, j \neq a}^{n+1} p_j\} \\
&= \emptyset
\end{aligned}
$$

Table 3.14: Node Addition Axiom 2 Proof

primary application instance. The second part of the proof is trivialized by the fact that axiom 2 was true at the beginning of the operation. Let P_i and P_j be any two of the previously existing nodes. The proof is shown in table 3.14.

Axiom 3 requires us to show that the document collection stored in each primary application instance is replicated across all remote application instances. Again, we will show this proof in two steps. First we prove it is true for any one of the previously existing primary application instances. We then show it is true for the newly added primary application instance. Let P_i be one of the previously existing application instances. P_a is the new primary application instance. The proof is shown in table 3.15.

Axiom 4 requires us to show that for each node, the document collection stored in each of the remote application instances are disjoint. We prove this axiom in three parts. First, we prove it is true for one of the previously existing nodes. Two steps are required for this portion of the proof. The first step shows it is true for any two of the previously existing remote application instances. The second step shows it is true for any one of the previously existing remote application instances and the new remote application instance.

$$
\begin{aligned}
\bigcup_{j=1,j\neq i}^{n+1} D'(S_{ji}) &= (\bigcup_{j=1,j\neq i,j\neq a}^{n+1} D'(S_{ji}))\bigcup D'(S_{ai}) \\
&= (\bigcup_{j=1,j\neq i,j\neq a}^{n+1} D(S_{ji}) - p_{ji} - r_{ji})\bigcup r_i \\
&= (\bigcup_{j=1,j\neq i,j\neq a}^{n+1} D(S_{ji}) - p_{ji} - r_{ji})\bigcup(\bigcup_{j=1,j\neq i,j\neq a}^{n+1} r_{ji}) \\
&= (\bigcup_{j=1,j\neq i,j\neq a}^{n+1} D(S_{ji}) - p_{ji} - r_{ji})\bigcup(\bigcup_{j=1,j\neq i,j\neq a}^{n+1} r_{ji}) \\
&= \bigcup_{j=1,j\neq i,j\neq a}^{n+1} D(S_{ji}) - p_{ji} \\
&= (\bigcup_{j=1,j\neq i,j\neq a}^{n+1} D(S_{ji})) - p_i \\
&= D(P_i) - p_i \\
&= D'(P_i)
\end{aligned}
$$

$$
\begin{aligned}
\bigcup_{j=1,j\neq a}^{n+1} D'(S_{ja}) &= \bigcup_{j=1,j\neq a}^{n+1} p_j \\
&= D'(P_a)
\end{aligned}
$$

Table 3.15: Node Addition Axiom 3 Proof

The second part of the proof shows that axiom 4 is true for node N_a. The proof for the new node depends entirely on the fact that axiom 2 was true at the beginning of the operation. Let N_i be any one of the previously existing nodes. Let S_{ji} and S_{ki} be any two previously existing remote application instances. The proof is shown in table 3.16.

$$\begin{aligned} D'(S_{ji}) \bigcap D'(S_{ki}) &= \{D(S_{ji}) - r_{ji} - p_{ji}\} \bigcap \{D(S_{ki} - r_{ki} - p_{ki}\} \\ &= D(S_{ji}) \bigcap D(S_{ki}) \\ &= \emptyset \end{aligned}$$

$$\begin{aligned} D'(S_{ji} \bigcap D'(S_{ai}) &= \{D(S_{ji} - r_{ji} - p_{ji}\} \bigcap r_i \\ &= \emptyset \end{aligned}$$

$$\begin{aligned} D'(S_{ia}) \bigcap D'(S_{ja}) &= p_i \bigcap p_j \\ &= \emptyset \end{aligned}$$

Table 3.16: Node Addition Axiom 4 Proof

Subset Selection

Unlike the node removal step, no data movement is required to take place to maintain a valid RDDS. The addition of a new node introduces more available resources, but, there is nothing in the RDDS definition that fources us to make use of them. Performing no data movements would be equivalent to selection empty subsets. Since it is our goal to have a balanced distribution whereever possible, our subsets of chosen such that a balanced distribution is maintained. We do not create a balanced distribution if one did not exist at the commencement of the operation, but our subset selections do not destroy a balanced distribution if it exists.

One major difference which makes the subset selection for a node addition operation more difficult than a node removal operation is that during a node addition operation, documents are added to the new application instances and removed from the previoulsly existing application instances. Since document deletions occur, care must be taken so that the subsets chosen to not incur a disproportionate number of document deletions from one or more of the previously existing application instances.

In step one of the node addition process, when selecting the subset r_c from $D(P_c)$ that will be stored in the new remote application instance S_{ac}, it is important to choose the subset such that each of the previously existing remote application instances are replicating a similar number documents from the subset r_c. Instead of choosing the subset r_c by looking at the document collection $D(P_c)$, it is easier to look at the document collections $D(S_{ic}), 1 \leq i \leq n+1, i \neq c, i \neq a$ and selecting the subsets r_{ic} individually and then calculating r_c by perform the set union of the individual r_{ic} subsets. The size of the subset r_{ic} chosen from $D(S_{ic})$ is roughly $1/n+1$ of its size at the beginning of the operation.

During step two, the subset p_c is calculated in a similar manner just described. Care must be taken so that $p_c \bigcap r_c = \emptyset$. Since we look at each of the remote application instances individually, we can enforce the disjoint requirement of p_c and r_c by ensuring that $p_{ic} \bigcap r_{ic} = \emptyset$ for each remote application instances.

Again, we give a brief explanation show that our subset selection algorithms maintain a balanced distribution assuming a balanced distribution previously existed. Let use assume that our global documetn collection consists of d documents. Table 3.9 shows the estimated document collection sizes for the application instances in a balanced valid RDDS distribution.

In step one, the document collection r_c is computed by combining the $n-1$ subsets r_{ic}. Each subset r_{ic} is chosen to be $1/n+1$ of the document collection $D(S_{ic})$. The size of the document collection $D(S_{ic})$ is $d/n(n-1)$ making each subset r_{ic} approximately $d/n(n-1)(n+1)$ in size and r_{ic} approximately $d/n(n+1)$. Using the same logic, we show that in step two, the subsets p_{ic} are approximiately $d/n(n-1)(n+1)$ in size and the subset p_c is approximatesly $d/n(n+1)$ in size.

Just like the analysis of the node removal process, we now show that if a balanced distribution exists before a node addition operation, the subsets we choose maintain a balanced distribution. Tables 3.17, 3.18 and 3.19 show the new sizes of the previously existing primary and secondary application instances. Tables 3.20 and 3.21 show the same for the newly added node. The estimated sizes shown correspond to a balanced distribution for a RDSS system with $n+1$ nodes.

$$\begin{aligned} |D'(P_i)| &= |D(P_i)| - |p_i| \\ &= d/n - d/n(n+1) \\ &= d/n+1 \end{aligned}$$

Table 3.17: Primary Application Size (Existing Node)

$$
\begin{aligned}
|D'(S_{ji})| &= |D(S_{ji}) - |r_i| - |p_i| \\
&= d/n(n-1) - d/n(n-1)(n+1) - d/n(n-1)(n+1) \\
&= d/n(n+1)
\end{aligned}
$$

Table 3.18: Remote Secondary Application Size (Existing Node)

$$
\begin{aligned}
|D'(S_{ai})| &= |r_i| \\
&= |\bigcup_{j=1, j\neq a, j\neq i}^{n+1} r_{ji}| \\
&= \sum_{j=1, j\neq a, j\neq i}^{n+1} |r_{ji}| \\
&= \sum_{j=1, j\neq a, j\neq i}^{n+1} d/n(n-1)(n+1) \\
&= d/n(n+1)
\end{aligned}
$$

Table 3.19: Foreign Secondary Application Size (Existing Node)

$$
\begin{aligned}
|D'(P_a)| &= |\bigcup_{i=1, i\neq a}^{n+1} p_i| \\
&= \sum_{i=1, i\neq a}^{n+1} |P_i| \\
&= \sum_{i=1, i\neq a}^{n+1} d/n(n+1) \\
&= d/n+1
\end{aligned}
$$

Table 3.20: Primary Application Size (Node Node)

$$
\begin{aligned}
|D'(S_{ia})| &= |p_i| \\
&= |\bigcup_{j=1, j\neq a, j\neq i}^{n+1} p_{ji}| \\
&= \sum_{j=1, j\neq a, j\neq i}^{n+1} |p_{ji}| \\
&= \sum_{j=1, j\neq a, j\neq i}^{n+1} d/n(n-1)(n+1) \\
&= d/n(n+1)
\end{aligned}
$$

Table 3.21: Secondary Application Size (New Node)

3.4.3 Document Collection Updates

In this section, we discuss the steps necessary for performing document collection updates in the RDSS. The two document collection operations allowed are to add a new document and to remove a previously existing document. In the RDSS, there are two types of document collection updates. A *global document collection update* is an update where the collection being modified is the entire global document collection. These types of updates apply to all primary application instances in the RDSS and hence all secondary application instances as well. A *primary document collection* update is an update that applies to a specific primary application instance and its corresponding remote secondary application instances.

The main reason for the separation of these two types of updates is simplicity. We will see that a global document collection update is in fact implemented by using the primary document collection update service. The primary document collection update service is required to ensure that axioms 3 and 4 of the RDDS definition are true at the termination of the operation. The global document collection update service enforces axioms 1 and 2 of the RDDS definition. By separating these two services, each service becomes simpler than it would otherwise have been.

Finally, the protocol for performing a global or primary document collection update modeled after the update protocol used by the application. In fact, update protocol used for performing global and primary document collection updates differs in only a few instances. This design was chosen so any client program written to perform document collection updates to a single application instances requires only minor modifications to allow it to perform global or primary document collection updates.

Primary Document Collection Updates

As mentioned above, performing updates to the primary document collection affects only a single primary application instance and all of its remote secondary application instances. When adding documents, the distribution server adds each document to the primary application instance and distributes each document to the remote secondary application instances on a round robin basis. When removing documents, each document is removed from the primary document collection and also removed from the appropriate remote secondary application instance. The main objective of the primary document collection update is to evenly distribute document additions over all remote secondary application instances. We use the fact that primary document collection updates are evenly distributed to prove that global document collection updates are also evenly distributed over all nodes in the distributed group.

A brief discussion of the four axioms of the RDDS definition is warranted. The first two axioms of the RDDS definition are not relevant since the primary document collection update is requires to enforce axioms 3 and 4 only. Axiom 3 states that the primary document collection must be replicated across all remote application instances. Since each incoming document is added to both the primary application instances and one of the remote application instances, axiom 3 is maintained. Furthermore, the fact that each incoming document is replicated by only one remote application instances, axiom 4 also hold true.

By distributing documents to the remote application instances in a round robin manner, we are also assured of a balanced distribution of the newly added documents. Provided we began with a balanced distribution of the primary document collection across all remote application instances, the new primary document collection will also be evenly distributed. The only situation that can cause the distribution to become unbalanced, after beginning with an balanced distribution is for a large number of documents to be deleted that are all replicated on a single remote application instance. While we could have tried to alleviate this problem, it is not completely avoidable without first reading all updates to be performed first. The latter option may not be feasible for large updates.

Figure 3.10 shows an example of a primary document collection update. The external client is performing an update to the primary document collection being stored in the primary application instance on node 1. When the client connects to the distribution server, the distribution server opens connections to the primary application instance and each of the remote secondary application instances. Once all connections have been established, the client proceeds with an update following the update protocol. When the client is finished performing the necessary updates, the updates have been applied such that a valid RDDS exists for the primary document collection being modified. The client has

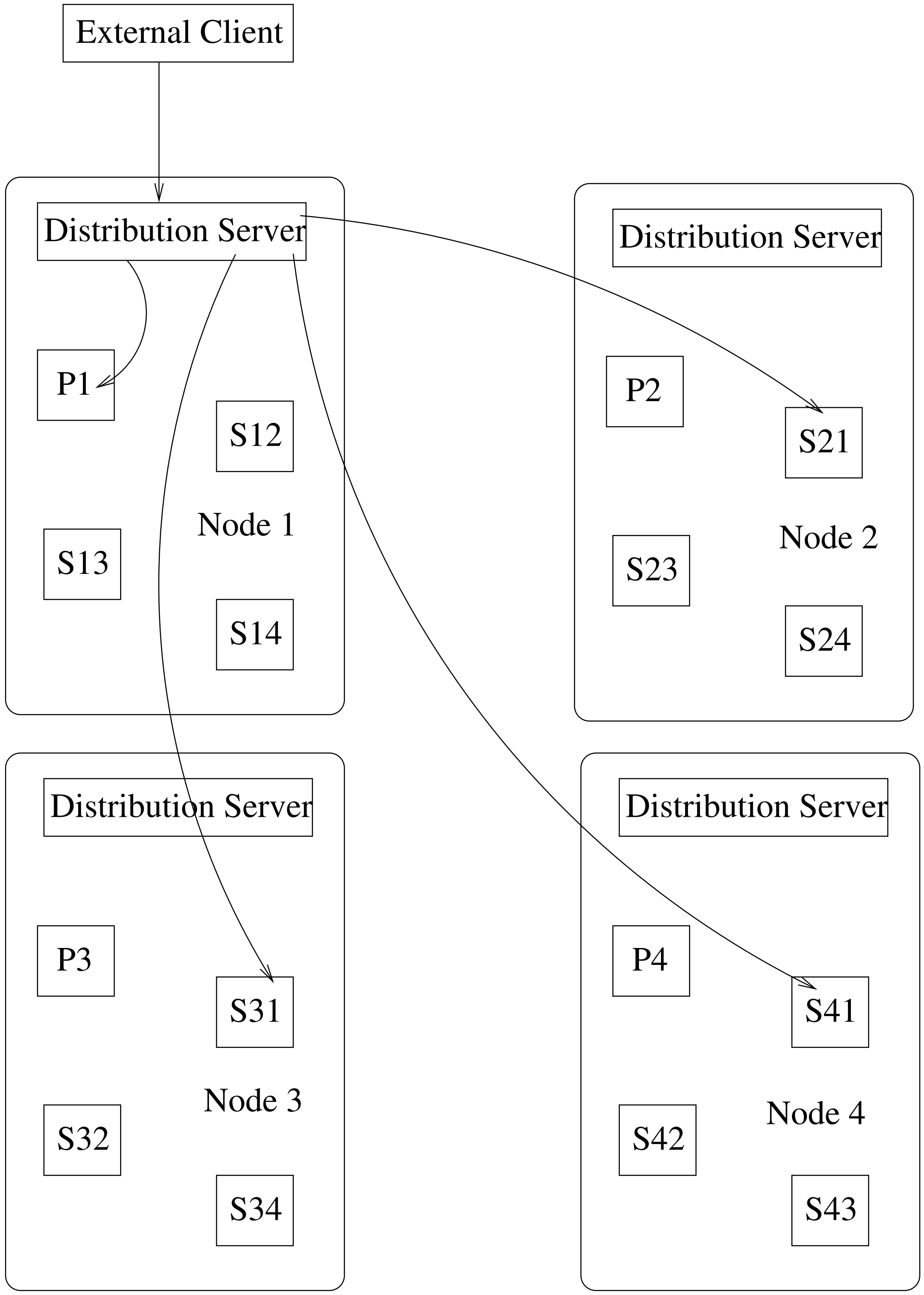

Figure 3.10: Primary Document Collection Update

done no work to ensure that a valid RDDS is in place at the completion of the update.

Global Document Collection Updates

We implement a global document collection update by making use of the primary document collection update services. Essentially, when an external client requests a global document collection update, the distribution server will turn around and initiate a primary document collection update with all distribution servers, including itself. The global document collection update is then distributed to all primary document collection updates in the same manner as primary document collection updates are performed. Document additions are handed out to each of the primary document collection updates in a round robin manner. Document deletions are sent to the distribution server performing the primary document collection update whose primary application instance is storing the document to be deleted.

Axiom 1 and 2 of the RDDS definition are maintained by the global document collection update service for the same reasons that axioms 3 and 4 are maintained by the primary document collection update service. Also, the round-robin system used for assigning replication duties gives us a balanced distribution of all new documents added to the system. Provided we began with a balanced distribution of the global document collection, we will be left with an even distribution at the end of the operation. The only exception to this claim requires an unfortunate set of document deletions.

Figure 3.11 shows the first step in performing a global document collection update. At this point, the client has connected to the distribution server on node 1 and indicated that a global document collection update is to be performed. Although the figure shows two distribution servers on node 1, there is in fact only one. The reason will be explained in the next step.

In figure 3.12, we see that after being told by the external client that a global document collection is to be performed, the distribution server opened a connection to each of the distribution servers in the distributed group, including itself. Each of these connections is opened to perform a primary document collection update, not a global document collection update. The reason for the appearance of a second instance of the distribution server on node 1 is to draw attention to the fact that the distribution server has, in fact, opened a connection to itself in order to perform the primary document collection update. At this point, the external client holds a connection to the first instance of the distribution server on node 1. The first instance of the distribution server holds a connection to the second instance of itself in order to perform the primary document collection update.

Finally, in figure 3.13, we see how all of the distribution servers performing primary document collection updates have opened connections to all necessary primary application instances and remote secondary application instances. The global document collection

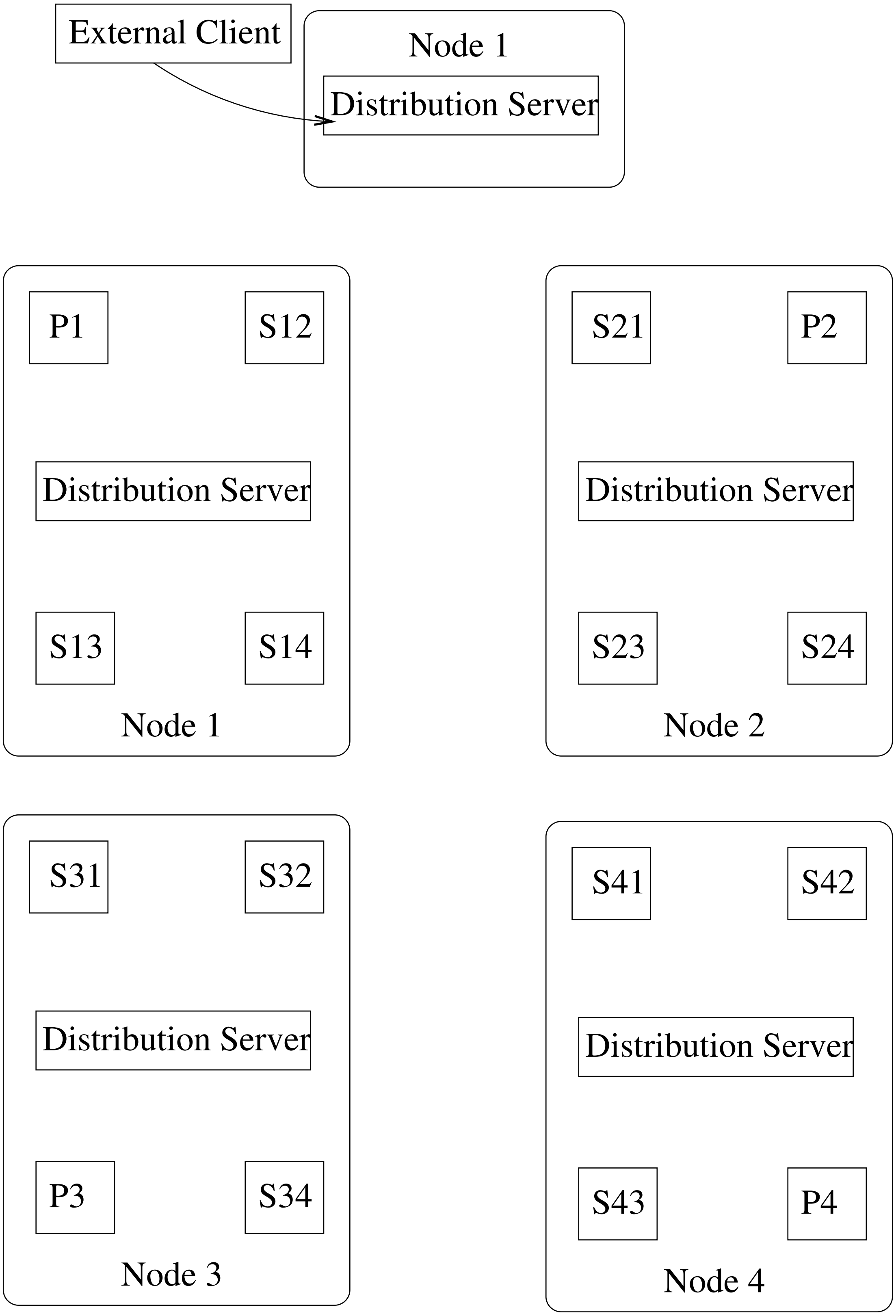

Figure 3.11: Global Document Collection Update, Step 1

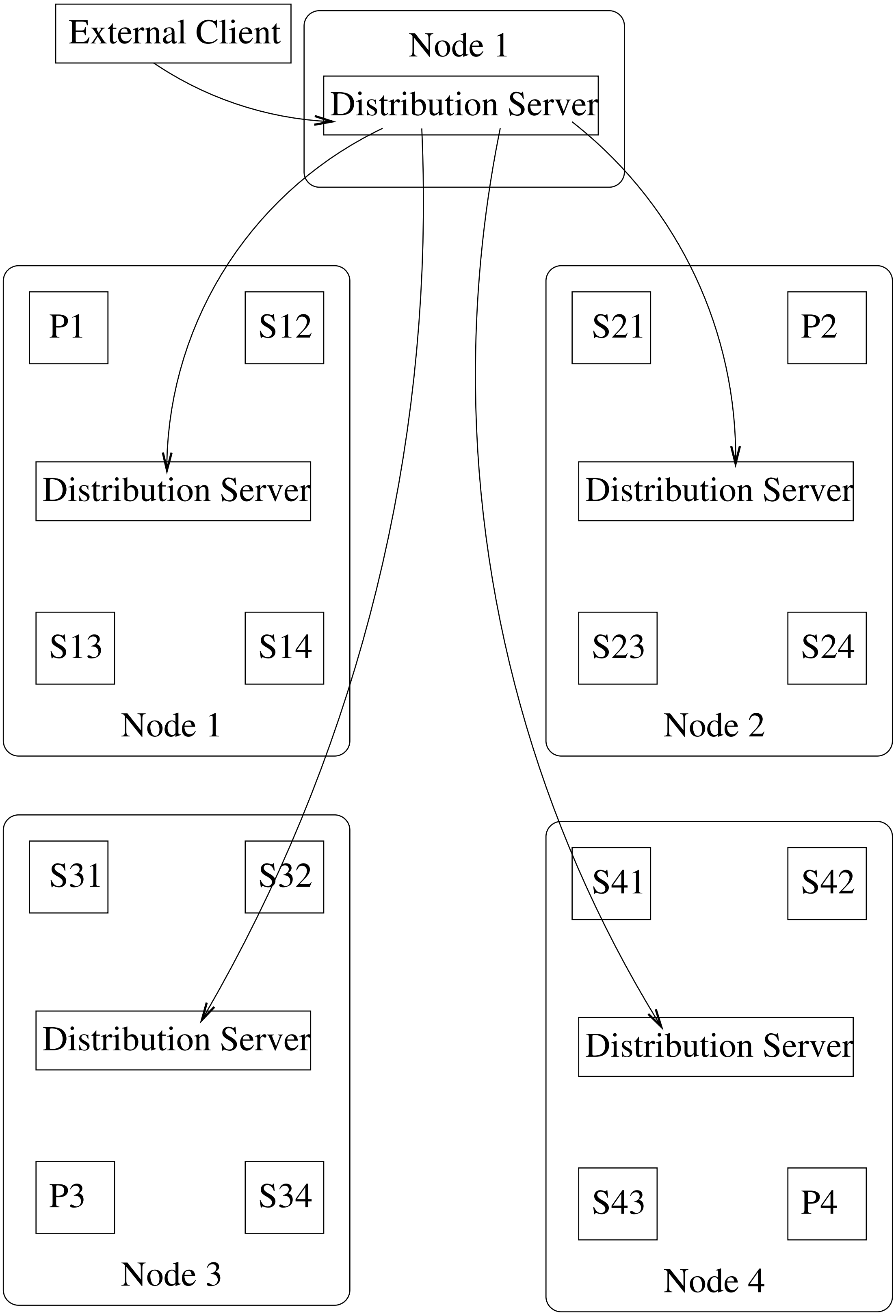

Figure 3.12: Global Document Collection Update, Step 2

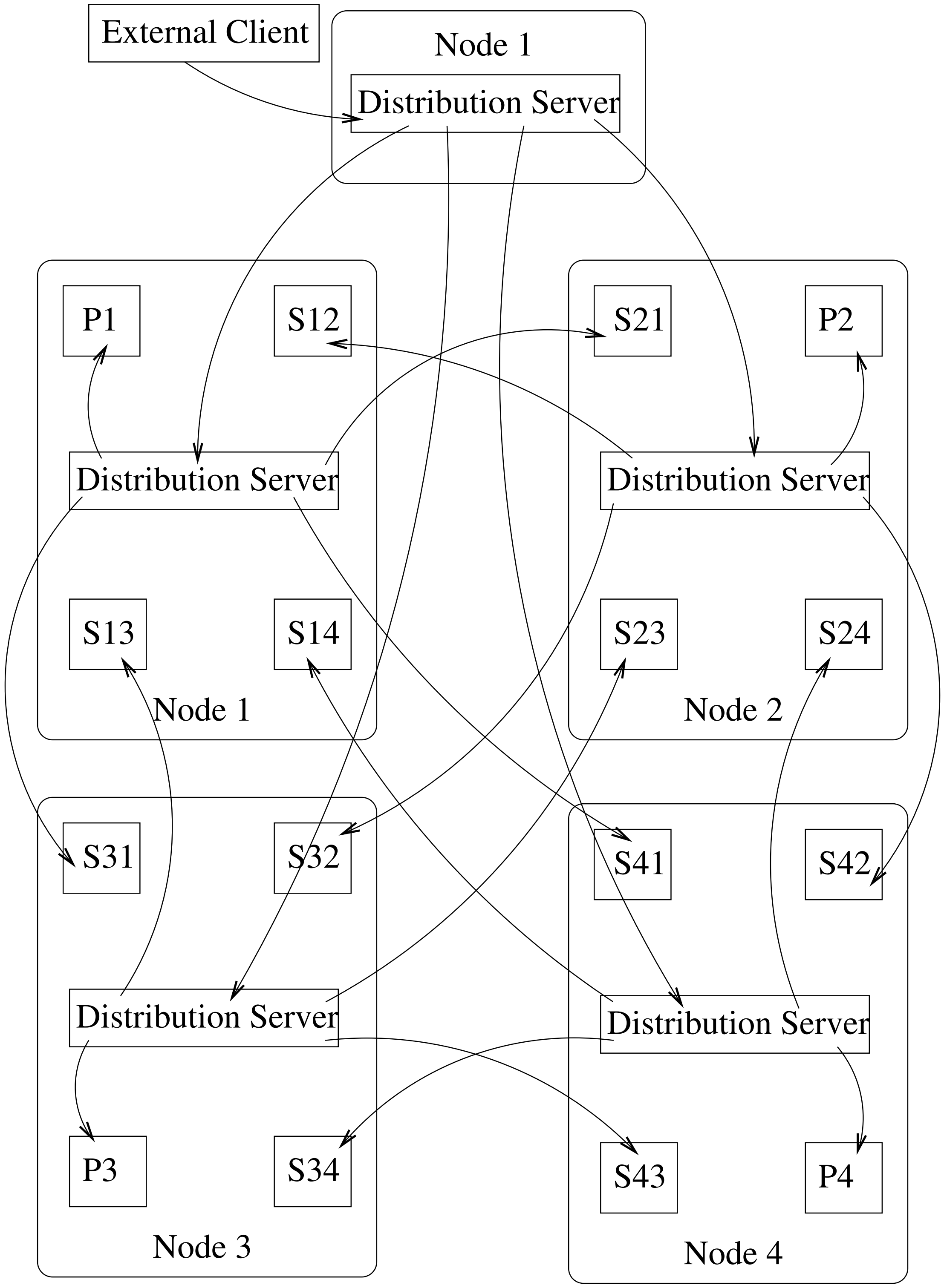

Figure 3.13: Global Document Collection Update, Step 3

update is now ready to proceed.

Chapter 4

Application Protocols

There are several operations that the application must support. These operations can be divided in to two main categories. The first category contains all commands related to performing a transaction, while the second category contains commands related to performing document collection updates. The protocols described below are by external clients wishing to perform updates to the document collection stored in an application.

Typically, each command issued to the application consists of a four byte command identifier. For each command the application receives, the application must return a four byte response message. Each response message is a four byte response identifier. Currently, the two possible response messages are `APP_SUCCESS` or `APP_ERROR`. Valid command identifiers are shown in table 4.1

Update Commands	**Transaction Commands**
`APP_LISTING`	`APP_BEGIN`
`APP_ADD`	`APP_PREPARE_EXPORTS`
`APP_DELETE`	`APP_PREPARE`
`APP_EXPORT`	`APP_COMMIT`
`APP_IMPORT`	`APP_ABORT`
`APP_EXPORT_DELETE`	

Table 4.1: Application Protocol Commands

With the exception of the `APP_LISTING` command, all of the commands listed in table 4.1 are valid for both the application and the application proxy. The `APP_LISTING` command is valid only for the application proxy. Additionally, before any update commands can be issued, a transaction must first be initiated. Figure 4.1 shows a finite automata describing the valid sequences of commands that can be sent to the application. It is an error to issue any command at a given state that does not have a transition leaving that state associated with the command.

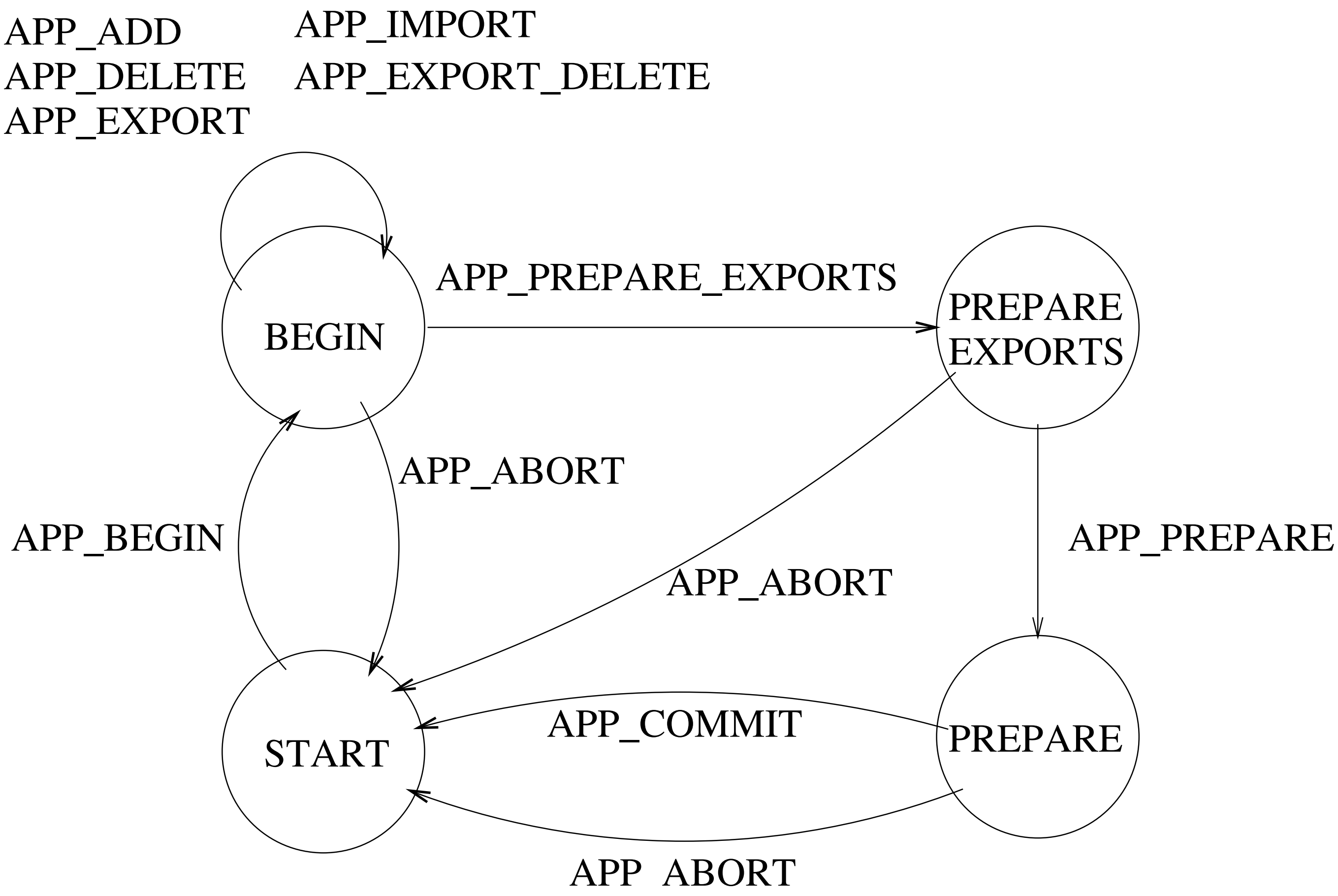

Figure 4.1: Update Protocol Diagram

4.1 Known/Unknown Protocols

Throughout the RDSS, there are multiple times in which a list of document headers, and possibly their underlying documents, must be sent from one component to another. Depending on the situation, it may or may not be known in advance how many documents are to be transmitted. In order for two components to successfully communicate with each other, the component reading the list of documents must either know in advance how many documents are to be sent, or, it must know when the list of documents being sent has ended. We have developed to different protocols for handling this situation. There is a `DIST_PROTO_KNOWN` protocol and a `DIST_PROTO_UNKNOWN` protocol. Suppose component S is to send a list of documents to component R. Below, we describe exactly what is transmitted by S, and how R knows what to expect. We assume that a connection already exists between S and R and the two components are ready to exchange the list of documents with each other.

4.1.1 Known Protocol

Component S wants to transmit n documents to component R. S begins by sending the four byte protocol identifier `DIST_PROTO_KNOWN`. The next four bytes contain the number of documents to be transmitted (n). On the other end, R reads the first four bytes and determines what protocol is being used to transmit the document list. After detecting that the `DIST_PROTO_KNOWN` protocol is being used, R now knows that the next four bytes contains the number of documents to be read.

Now that a common protocol has been established between S and R, the data exchange can now take place. S simply writes out each of the document headers, one after another with no separation marker in between. Since document headers are fixed in length, an implicit separation marker exists. If S and R also want to exchange the document data as well, S and R can agree, in advance, of how to send the data. The solution used in our prototype system was to include the document length in the document header. Immediately after sending a document header, the raw document data was then transmitted. Other alternatives exist and no restriction is placed between S and R by the `DIST_PROTO_KNOWN` protocol. When all document headers and possibly the document data has been sent, S simply closes its connection. It is the task of R to detect transmission errors and report such errors to its controlling component. Even though S does not detect the error, R will, and ultimately, any transaction involving the failed communication will be aborted.

`DIST_PROTO_KNOWN`	4 bytes
`NUM_DOCS`	4 bytes
`DOCUMENT_HEADER_1`	$fixedlength$ bytes
`DOCUMENT_HEADER_2`	$fixedlength$ bytes
⋮	⋮
`DOCUMENT_HEADER_n`	$fixedlength$ bytes

Table 4.2: Known Protocol, Document Headers only

4.1.2 Unknown Protocol

Component S wants to transmit an unknown number of documents to R. The protocol is similar to the known protocol except that we must now explicitly write out separation markers between documents. Component S begins by writing out the four byte protocol

`DIST_PROTO_KNOWN`	4 bytes
`NUM_DOCS`	4 bytes
`DOCUMENT_HEADER_1`	$fixedlength$ bytes
`DOCUMENT_DATA_1`	$variablelength$ bytes
`DOCUMENT_HEADER_2`	$fixedlength$ bytes
`DOCUMENT_DATA_2`	$variablelength$ bytes
⋮	⋮
`DOCUMENT_HEADER_n`	$fixedlength$ bytes
`DOCUMENT_DATA_n`	$variablelength$ bytes

Table 4.3: Known Protocol, Document Headers and Data

identifier `DIST_PROTO_UNKNOWN`. Component R reads the first four bytes and determines that the unknown protocol is to be used.

Now that a common protocol has been established between S and R, the data exchange can now take place. Before every document that S writes out, S must first write out the four byte separator `DIST_PROTO_MORE`. After the separator has been written, S then writes out the document header and the optional document data just as it would in the known protocol. After writing out the separator, the document header and the document data, S has completely written out one document and can proceed to write out the next document be repeating the above steps. After S finally decides that it has no more documents to send, S terminates the data exchange by writing out the four byte terminating separator `DIST_PROTO_END`. After the terminating separator has been written, S closes its connection to R. From R's perspective, when it sees that the unknown protocol is to be used, R immediately tries to read the four byte separator. If the separator is `DIST_PROTO_MORE`, R knows that a document header, or both the header and data follow. After reading the document header and possibly the document data, R will attempt to read the next four byte separator. The process is repeated until R sees the `DIST_PROTO_END` separator and knows that there are no more documents. Just like the known protocol, it is up to R to detect transmission errors and report such errors to its controller component.

4.2 Document Collection API

As per figure 4.1, no update commands can be issued until a transaction has been initiated and the application is in the $BEGIN$ state. Once in the $BEGIN$ state, there is no limit on the number, or combination of update commands that can be given to the application for a particular transaction. Multiple instances of the same command are acceptable,

`DIST_PROTO_UNKNOWN`	4 bytes
`DIST_PROTO_MORE`	4 bytes
`DOCUMENT_HEADER_1`	*fixedlength* bytes
`DIST_PROTO_MORE`	4 bytes
`DOCUMENT_HEADER_2`	*fixedlength* bytes
⋮	⋮
`DIST_PROTO_MORE`	4 bytes
`DOCUMENT_HEADER_n`	*fixedlength* bytes
`DIST_PROTO_END`	4 bytes

Table 4.4: Unknown Protocol, Document Headers only

`DIST_PROTO_UNKNOWN`	4 bytes
`DIST_PROTO_MORE`	4 bytes
`DOCUMENT_HEADER_1`	*fixedlength* bytes
`DOCUMENT_DATA_1`	*variablelength* bytes
`DIST_PROTO_MORE`	4 bytes
`DOCUMENT_HEADER_2`	*fixedlength* bytes
`DOCUMENT_DATA_2`	*variablelength* bytes
⋮	⋮
`DIST_PROTO_MORE`	4 bytes
`DOCUMENT_HEADER_n`	*fixedlength* bytes
`DOCUMENT_DATA_n`	*variablelength* bytes
`DIST_PROTO_END`	4 bytes

Table 4.5: Unknown Protocol, Document Headers and Data

even though it might turn out that those operations may conflict with each other when preparing the transaction. An example of a conflicting operation would be to add the same document more than once. This is a conflict because we do not know if the same document has been added twice or if two different documents have been assigned the same document identifier. In the former case, the second copy could be discarded, in the latter case, an error has occurred and should be reported. Furthermore, there is no requirement that each command be given to the application sequentially. The ramifications of this flexibility is that the application must support multiple concurrent update clients for all of the update commands.

While there are no restrictions on the number or types of commands, there are limitations on the order in which documents are supplied to the application. Any list of documents must be given to the application in sorted order according to the document number. The reason for this is performance. In order for the overhead of the application proxy to be minimal, a co-sequential scan is performed allowing it to modify its list of documents stored in the application in an efficient manner.

4.2.1 APP_LISTING

This command is valid only for the application proxy. The application proxy is responsible for maintaining information about each document being stored in the underlying application. The command is used primarily by the distribution server when it needs to discover what documents reside in an application so it can make a decision about moving, deleting or adding new documents to the application. By issuing the `APP_LISTING` command, an external client is returned a list of document headers corresponding to the collection of documents being stored in the underlying application. This command can be issued to the application instance at any time. Below is a step by step outline of a proper request/response cycle.

```
Step 1:  read 4 byte APP_LISTING command
Step 2:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
Step 3:  If success response written in step 2, write out all document
headers according to the known or unknown protocols
```

Figure 4.2: APP_LISTING protocol

4.2.2 APP_ADD

This command is used when adding new documents to the document collection. The operation can be used to add a single document, or, it can be used to batch up hundreds or even thousands of documents in one operation. The client must supply a document header and all document data to the application for each document to be added to the document collection. The only response that is needed from the application is an indication of the success or failure of the reading and storing of all pertinent information regarding the addition of the requested documents.

```
Step 1:  read 4 byte APP_ADD command
Step 2:  read all document headers and document data according to the
known or unknown protocols
Step 3:  If write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.3: APP_ADD protocol

4.2.3 APP_DELETE

This command is used for removing documents from a document collection. The client must supply the document header for each document that is to be deleted. The document header contains the document the unique document identifier. It is this information that is used to identify which documents should be deleted when preparing the transaction. The only response that is needed from the application is an indication of the success or failure of the reading and storing of all pertinent information regarding the deletion of the requested documents.

```
Step 1:  read 4 byte APP_ADD command
Step 2:  read all document headers according to the known or unknown
protocols
Step 3:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.4: APP_DELETE protocol

4.2.4 APP_EXPORT

When migrating documents from one application instance to another, it is sometimes not possible to reconstruct the original documents. An example could be an application that has already passed the original data through an irreversible filter when the original data was added to the document collection. Even if it were possible to reconstruct the original data, it might be easier to allow the application instances to transfer the documents using some internal format. The `APP_EXPORT` allows an application to make available a set of documents to be transferred in the application's internal format. The application must read the all document headers describing what documents are to be transferred. Furthermore, the application must establish a TCP server port to accept an incoming connection. It is through the TCP server port that the other application will make contact and allow the two applications involved to transfer the necessary documents. The response from the application consists the TCP port number to be used for establishing contact and an indication of the success or failure of the reading and storing of all pertinent information regarding the exporting of the specified documents.

```
Step 1:  read 4 byte APP_EXPORT command
Step 2:  write out 2 byte export port
Step 3:  read all document headers according to the known or unknown
protocols
Step 4:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.5: APP_EXPORT protocol

4.2.5 APP_IMPORT

The APP_IMPORT operation is the complementary operation to the APP_EXPORT operation. The application will first read the exporter application's connection information and then read all document headers of the documents to be imported. The response from the application is an indication of the success or failure of reading and storing all pertinent information regarding the importing of the specified documents.

4.2.6 APP_EXPORT_DELETE

This operation is identical to the APP_EXPORT operation except that the documents specified should also be deleted after they are exported.

```
Step 1:  read 4 byte APP_IMPORT command
Step 2:  read 4 byte IP address of exporter
Step 3:  read 2 byte TCP port of exporter
Step 4:  read all document headers according to the known or unknown
protocols
Step 5:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.6: APP_IMPORT protocol

```
Step 1:  read 4 byte APP_EXPORT_DELETE command
Step 2:  write out 2 byte export port
Step 3:  read all document headers according to the known or unknown
protocols
Step 4:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.7: APP_EXPORT_DELETE protocol

4.3 Transaction API

The transaction API that must be supported by the application is a slight variation of the standard transaction operations. While, in general, the application must support multiple simultaneous client query connections, the application can be guaranteed that there will be at most one update client issuing transaction commands to the application. More specifically, if the application is busy performing work associated with one of the transaction commands, the application is guaranteed not to have a second transaction command issued until the first one has finished. This simplifies some concurrency issues regarding the transaction progression. The transaction commands are listed in table 4.1

4.3.1 APP_BEGIN

This command is sent to the application to begin a new transaction. After this command, the application must be prepared to accept work that must be performed as part of the transaction. The response from the application is an indication of the success or failure of the operation.

```
Step 1:  read 4 byte APP_BEGIN command
Step 2:  perform any work required to begin a new transaction
Step 3:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.8: APP_BEGIN protocol

4.3.2 APP_PREPARE_EXPORTS

Essentially, we have been forced to break the prepare statement of a transaction into two commands. This was due to a race condition that existed between exporters and importers. With only one prepare statement, every application instance will attempt to prepare the work assigned to it as part of the transaction. For every exporter, there is an importer. The race condition is that the importer may try to establish a connection with the exporter before the exporter is ready. Our solution was to separately prepare the exporters, wait for confirmation that they are all ready and then prepare the rest of the transaction. When the command is received, the application must perform any necessary work so that the application is ready for any importer attempting to establish a connection. The response from the application is an indication of the success or failure of the operation.

```
Step 1:  read 4 byte APP_PREPARE_EXPORTS command
Step 2:  perform any work required
Step 3:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.9: APP_PREPARE_EXPORTS protocol

4.3.3 APP_PREPARE

This command is the second part of the prepare statement that we were forced to break into two parts. At this point, the application is instructed to perform whatever work is necessary to prepare the transaction. These tasks are entirely application dependent, but, at completion of all work, the new document collection should have all documents of the previously existing document collection plus any new documents being added or imported. The data for the added documents should already exist on storage since it was given to the application when the document collection command APP_ADD was given. The data for all documents being imported will be read from the corresponding exporting application in real time, when the transaction is being prepared. When all work has been completed, the application must indicate the success or failure of the command be return an appropriate return code.

```
Step 1:  read 4 byte APP_PREPARE command
Step 2:  perform any work required
Step 3:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.10: APP_PREPARE protocol

4.3.4 APP_ABORT

At any time during a transaction, the transaction can be aborted for a number of reasons. An error from any application can cause the distributed transaction to be aborted. As a result, it is possible that when an application is told to abort the current transaction, the application could be in any number of states regarding its own readiness to commit the transaction. The application must take immediate action to abort the transaction.

```
Step 1:  read 4 byte APP_ABORT command
Step 2:  perform any work required
Step 3:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.11: APP_ABORT protocol

```
Step 1:  read 4 byte APP_COMMIT command
Step 2:  perform any work required
Step 3:  write out 4 byte response (APP_SUCCESS/APP_ERROR)
```

Figure 4.12: APP_COMMIT protocol

Although the application is required to abort the transaction, we do require the application to respond indicating the success or failure of the transaction abort operation. No further transactions can be attempted before confirmation is received that the current transaction is completely aborted. While we place no requirement on when or how long it takes to abort the transaction, the sooner the transaction is aborted the better.

4.3.5 APP_COMMIT

This command is issued only when the distributed consensus is that the transaction is to be committed. By definition, all applications voted to commit and so when an application receives this command, the application had at one point prepared all of its work for the transaction and was prepared to commit. The application must now perform any necessary work to commit the transaction. Although an application's decision to commit is irrevocable, because unexpected errors might occur, we do require the application to respond indicating its success or failure to commit the transaction.

Chapter 5

Implementation Details

In chapter 3, the design principles behind the RDSS was given. In this chapter, we go further into detail of component and give an implementation that fulfills the design requirements. We we examine the RDSS on a component-by-component basis.

5.1 IO Proxy

5.1.1 Overview

The IO Proxy is responsible for transferring data between a local component and a remote IO Proxy while simultaneously monitoring the local GSM controller for heartbeat messages. If at any time, the gsm indicates a lack of heartbeat messages from a remote node over a period predefined period of time, the IO Proxy deems the remote node to have potentially failed. Upon deeming a node to have potentially failed, all connections between local components and components on the remote node in question are severed.

For each virtual connection created between two components, three direction connections are established. A *direct* connection is a physical TCP connection between two components. In figure 5.1, a virtual connection between component A on node 192.168.0.25 and component B on node 192.168.0.26 is shown. The three direct connections are labelled C_1, C_2 and C_3. The IO Proxy on node 192.168.0.25 is responsible for transferring data between connections $C1$ and $C2$, while the IO proxy on node 192.168.0.26 is responsible for transferring data between connections $C2$ and $C3$. The association of two direct connections, whose data must be shuttled from one to the other, is called a *tunnel*. We use the verb *tunneling* to be the action of moving data between the two endpoints of a tunnel.

The IO Proxy is a single threaded process, and as a result, all IO performed within the IO Proxy is non-blocking IO. Since there is only one thread, non-blocking IO is required for reliability and performance. Performing blocking IO could result in the IO Proxy waiting

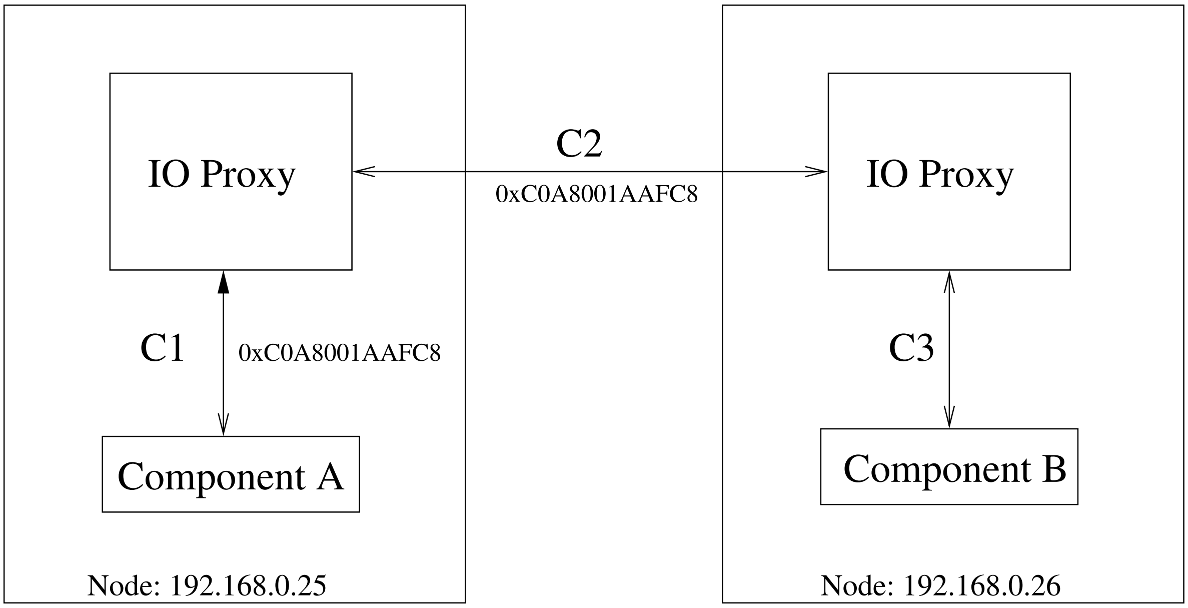

Figure 5.1: Two tunnels make up one virtual connection

for data to be read from, or written to, an endpoint even though data is available to be read from, or written to, one or many other endpoints belonging to different tunnels. Waiting for slower components hinders the performance of all components connected to the IO Proxy. Taken to the extreme, the performance argument just given can be used to show how a deadlock situation could occur. Without imposing some restrictions on the communication between components, there is no way of guaranteeing that a deadlock situation can not occur. Our implementation uses the Unix `select` statement to monitor multiple connections simultaneously. The nature of non-blocking IO is that a `write` operation may not write all the data that was requested. Additionally, one endpoint might write data faster than the other endpoint reads the data. As such, two small internal buffers are used to temporarily store data that has been read from one endpoint and not yet written to the other.

The core of the IO Proxy is a collection of finite automata. Each tunnel has a corresponding state, and with the exception of reading data from, and writing data to, various tunnel endpoints, the majority of the work performed by the IO Proxy is maintaining the state of each tunnel. The IO Proxy consists of an infinite loop with three main activities. The first activity is to check the state of all active tunnels and take the appropriate action. The state of the tunnel indicates if the tunnel is to be destroyed, or what endpoints of the tunnels can be read from, or written to. During the analysis of the state of each tunnel, the appropriate bits in the `fdset` structures are set for the `select` call. The second step is the issue of an infinite-blocking `select` call. This call will return only when there is work to be performed. Finally, the last step in the loop is to inspect the `fdset` structures from the `select` call, decide what actions need to take place and to do those actions. Table 5.1 shows a quick reference view of the algorithm used in the IO Proxy.

Begin Loop		
	Step 1	Tunnel State Analysis / Set up `fdset` structures
	Step 2	Perform blocking `select` call
	Step 3a	Accept new Connections/Tunnels
	Step 3b	Read/Write data
	Step 3c	Inspect GSM heartbeat
End Loop		

Table 5.1: IO Proxy Logic

5.1.2 Communication Channels

The IO Proxy listens for incoming connections on a well known TCP port. This port is the mechanism by which any RDSS component establishes an inter-node virtual connection. In order to receive heartbeat messages from the local GSM controller, the IO Proxy establishes a TCP connection with the local GSM controller. Every heartbeat message generated by the local GSM controller will be relayed to the IO proxy using this connection.

5.1.3 Tunneling

Additional notation is needed before proceeding with the details of a tunnel. For each virtual connection between two components, two tunnels are created. A tunnel is created in each of the two IO Proxys involved in the virtual connection. The two tunnels are created sequentially. The *initial* tunnel is the first of the two tunnels to be created and the *final* tunnel is the last of the two tunnels to be created.

Each tunnel in the IO Proxy consists of two endpoints, two buffers and a state, indicating what operations are currently valid for the tunnel. Endpoints are represented by Unix sockets. The buffer associated with each endpoint is used to store data that has been read from the endpoint, but which has not yet been written to the other endpoint. The state of the tunnel indicates if any errors have occurred, if there is data in the buffers to be written, or if the tunnel is still in the process of being established.

A tunnel is also established in stages, much like a virtual connection is established in stages. The connections to the two endpoints of a tunnel are also created sequentially. The endpoint whose connection is established first is the *source* endpoint while the other endpoint is the *final* endpoint. The source endpoint is the endpoint from which the IO Proxy accepted a connection. The final endpoint is the endpoint to which the IO Proxy established a connection.

5.1.4 Establishing a Tunnel

The first step in establishing a tunnel is for a component, local or remote, to establish a direction connection to the IO Proxy using the well known TCP port. After accepting the connection, the IO Proxy allocates a buffer to be used to store data that has been read from the source endpoint, and the tunnel is said to be in the `GET` state. The `GET` state signifies that the IO Proxy must read from the source endpoint the 6-byte destination address of the target endpoint. While in the `GET` state, the IO Proxy monitors the source endpoint for any incoming data. When data arrives, the IO Proxy adds it to the buffer until the 6-byte destination address has been read.

When the 6-byte destination address has been read, the IO Proxy creates a buffer for the target endpoint and proceeds to establish the connection to this endpoint. If the IP address of the target endpoint is the local node's IP address, then the local IO Proxy will establish a connection with the local component using the 2-byte TCP port that was sent as part of the 6-byte destination address. If the IP address of the target endpoint is a remote node, the local IO Proxy will establish a connection to the remote IO proxy for the target node using the well known TCP port. Since all IO is non-blocking, the connection to the target endpoint may or may not established immediately. If the connection cannot be established immediately, the tunnel is put into the `WAIT` state. In the `WAIT` state, the IO Proxy monitors the target endpoint to learn when the connection has been fully established. Eventually, the connection to the target endpoint will be established and the tunnel establishment can proceed to the next stage. In the event that the connection to the target endpoint was immediately established immediately, the `WAIT` state will be bypassed.

Once the IO Proxy has been notified that the connection to the target endpoint has been established, the IO Proxy must decide if the current tunnel is the initial or final tunnel in the virtual connection being established. If the target destination of the tunnel is a component of the local node, the current tunnel is the final tunnel of the virtual connection and no further action is required. The tunnel is set to the `CONNECTED` state and bidirectional communication can now take place between both endpoints of the tunnel. However, if the target destination is a connection to a remote IO Proxy, the current tunnel is the initial tunnel of the virtual connection and the IO Proxy must forward the target destination address to the remote IO Proxy. In this case, the tunnel is put in the `CONNECTING` state. While in the `CONNECTING` state, the local IO Proxy monitors the target endpoint and sends the 6-byte destination address when possible. After sending the 6 byte destination address to the remote IO proxy, the tunnel is put in the `CONNECTED` state and bidirectional communication can now take place.

Figure 5.2 shows all states associated with establishing a tunnel initiated by a local component. Figure 5.3 shows all states associated with establishing a tunnel initiated by a remote IO proxy.

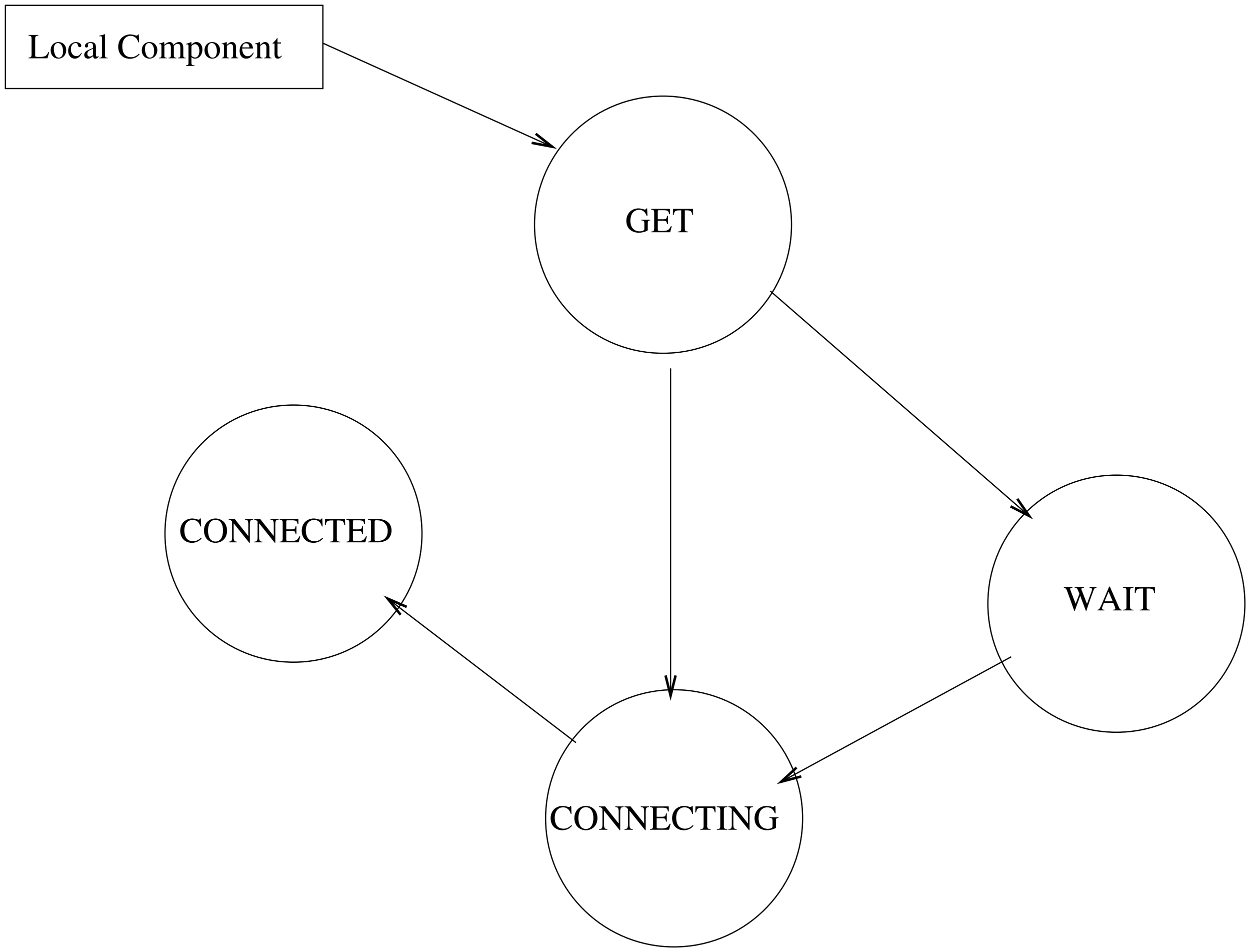

Figure 5.2: Tunnel establishment initiated by local component

5.1.5 Tunnel Forwarding

Once a tunnel is completely established, the finite automata of the tunnel remains in the `CONNECTED` state, although additional state is maintained to indicate when buffers are full or when errors occur. Recall that for each endpoint of a tunnel, there is an associated buffer that is used to store data read by the IO Proxy. For clarity sake, let us denote the two endpoints E_1 and E_2. The corresponding buffers will be called B_1 and B_2. It is important to note that the reading buffer for one endpoint is the writing buffer of the

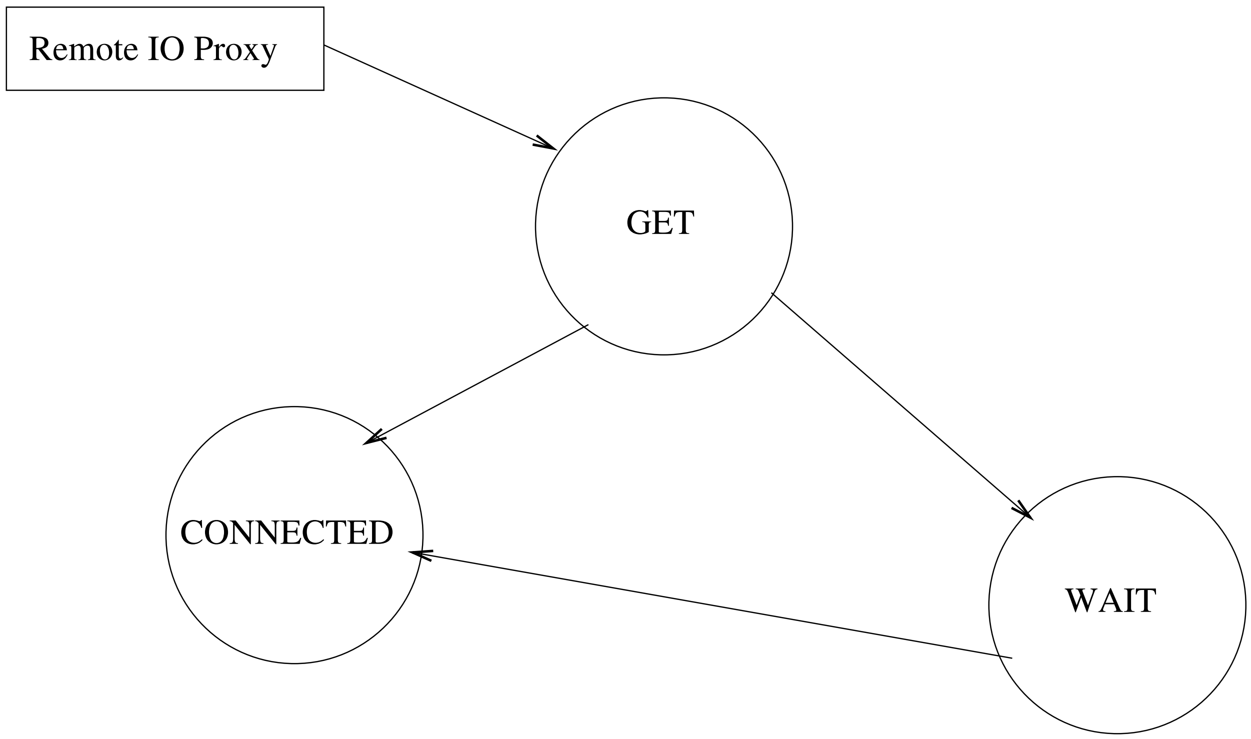

Figure 5.3: Tunnel establishment initiated by remote IO Proxy

other. Using our notation, any data read from endpoint E_2 will be stored in buffer B_2 and then written to E_1. Similarly, any data read from E_1 will be stored in B_1 and then written to E_2.

Whenever data is available to be read from an end point, the IO Proxy will read as much data as possible that will fit in the remaining space of the buffer. Once the buffer is full, the IO Proxy will not read any more data from the endpoint until there is room. Room is made in an endpoint's buffer when the IO Proxy writes some of the buffer to the other endpoint of the tunnel. The IO Proxy constantly manages the state of each tunnel so it can make two important decisions. The first decision it must make is to determine if there is any data currently stored in the internal buffers than can be written to an endpint. If so, the IO Proxy will monitor the endpoint's socket to see when data can be written. Secondly, the IO Proxy checks the read buffer of the endpoint and if it is not full, the IO Proxy will monitor for any incoming data from that endpoint's socket. These two decisions are made for each endpoint of every tunnel.

5.1.6 Timeout and Errors

If an IO error occurs when reading from, or writing to, an endpoint of a tunnel, the `ERROR` flag is added to the tunnel state. Similarly, if a node has been determined to have potentially failed, the `TIMEOUT` flag is added to each tunnel that contains an endpoint on the potentially failed node. During the first step of each iteration of the infinite loop, the IO Proxy inspects the state of all active tunnels. Any tunnel with an `ERROR` flag or a `TIMEOUT` flag is destroyed.

5.1.7 Tunnel Destruction

Tunnel destruction involves two tasks. First, the sockets of both endpoints of the tunnel must be closed. Additionally, the buffers associated with the tunnel are released.

There are two events that can cause the destruction of a tunnel. The first is for an error to occur, or for a remote node to be deemed to have potentially failed. Any data already read from one endpoint, but not yet written to the other is discarded. The second event that can cause the destruction of a tunnel is for one, or both, endpoints of a tunnel to close their connection to the IO Proxy. When an endpoint closes its connection, the `EOF` flag is set for that specific endpoint in the tunnel state. Any data already read from the closed endpoint is not discarded. The data will be written to the other endpoint in a timely manner. A tunnel can be destroyed when the connections to both endpoints of the tunnel have been closed or, one endpoint has been closed and there is no more data to be written to the other endpoint.

5.1.8 GSM Controller Monitoring

Monitoring the local GSM controller is simple. The IO Proxy monitors the connection to the GSM controller just as it monitors any other connection for activity. When the IO Proxy has read an entire heartbeat message, the IO Proxy proceeds to look for any potentially failed nodes. A threshold as been set in the IO Proxy that is used to determine when a node has potentially failed. The threshold is specified as a number of heartbeat messages that have been sent by the local GSM controller without having heard a single heartbeat message from a remote GSM controller. The IO Proxy compares the count of each node in the heartbeat message with the threshold value. If a node is found whose count exceeds the threshold value, the node is considered *potentially failed*. The IO Proxy scans all active tunnels, and any tunnel containing an endpoint on the potentially failed node is marked with the `TIMEOUT` flag. As per the timeout/error discussion above, the affected tunnels will be destroyed during the next iteration of the infinite loop.

5.2 GSM Controller

5.2.1 Overview

The GSM controller has two main tasks. The first task is running the GSM algorithm and the second is to allow external components to query the current state of the GSM. The nature of the GSM controller is very similar to that of the IO Proxy. The GSM controller must monitor activity from multiple IO sources concurrently. For the same reasons as the IO Proxy, the GSM controller also uses non-blocking IO for all operations.

Just like the IO Proxy, the GSM controller is a single threaded process which uses the UNIX `select` statement to monitor multiple file descriptors for activity. The same infinite loop that is present in the IO Proxy is also present in the GSM controller. There is one additional aspect to the GSM controller that is not present in the IO Proxy. The GSM controller must generate a new heartbeat message on a periodic basis. This is accomplished by setting an alarm. When the alarm "goes off", a signal handler is called to notify the GSM controller that it is time to generate a new heartbeat message.

5.2.2 Communication Channels

There are three points of contact for the GSM controller. The GSM controller listens for incoming TCP connections from query clients on a well known port. Any client wishing to inquire about the state of the GSM must contact the GSM on this port. The GSM controller also listens for incoming TCP connections from update clients on a different well known port. This port is used solely for the purpose of initiating an update operation within the distributed group.

The last point of contact for the GSM controller is exclusively for the use of the GSM controller itself and all remote GSM controllers. Each GSM controller broadcasts new heartbeat messages on, and listens for for incoming heartbeat messages from, a well known UDP port. Although this is really one communication channel, from an implementation standpoint, two file descriptors are needed: one for reading, the other for writing.

5.2.3 Running the GSM Algorithm

The GSM algorithm provides a limited API and it is through this API that the GSM Controller runs the GSM algorithm. The API is listed in table 5.2.

In order for the GSM controller to run the GSM algorithm, the GSM controller must have some working space on permanent storage to store the current GSM state. As the GSM algorithm transitions from state to state, the GSM algorithm stores the state of the GSM to permanent disk to allow it to recover after a node failure. The GSM algorithm,

API Call	Purpose
gsm_start	Restart the GSM after a node failure
gsm_initial	Start the GSM to initialize a new distributed group
gsm_empty	Start the GSM to be added to a previously existing distributed group
gsm_incoming	Pass an incoming heartbeat message to the GSM algorithm
gsm_tick	Generate a heartbeat message corresponding corresponding to the current state of the GSM
gsm_response	Provide feedback to the GSM regarding the success/failure of the current operation being performed
gsm_update	Request a global update operation to take place

Table 5.2: GSM Algorithm API

occasionally, but deterministically, stores the current GSM state to permanent disk. This stored state is essentially a snapshot of the state of the GSM in the event that a node failure occurs. If a node failure occurs, the snapshot will be used by the GSM algorithm as a starting point when recovering from a failure.

The GSM algorithm can be started in one of three ways. It can be stared when initializing a new distributed group, it can be started with the intent of being added to a previously existing distributed group or it can be started as the result of a node failure. When creating a new distributed group, the group contains only two nodes. The gsm_initial API call is used in each of the GSM controllers running on the nodes that will form the initial two members of the distributed group. Parameters to the gsm_initial call are the IP addresses of both nodes. When a new node is to be added to the distributed group, the GSM controller is started using the gsm_empty API call. This call requires the IP address of the local node. Finally, when a node fails and ultimately restarts, the gsm_start API call is used. Like the gsm_empty API call, the gsm_start call also requires the IP address of the local node. The difference between the gsm_start API call and the first two is that the gsm_start call uses the previously stored GSM state as a starting point.

Once initialized, incoming heartbeat messages from all GSM controllers are passed to the GSM algorithm as input by calling the gsm_incoming API. When the alarm goes off, the GSM controller calls the gsm_tick API to generate the heartbeat message to be sent to all remote GSM controllers. The majority of the work required to run the GSM algorithm involves calling the gsm_incoming and the gsm_tick API calls. From time to time, a state transition occurs in the GSM that requires external input before any further transitions can take place. When the GSM enters a PREPARE state, external actions are required to performed the operation being attempted. The GSM controller relays the success or failure of the attempted operation to the GSM algorithm by calling the gsm_response API. The last function `gsm_update` is used by the GSM controller to indicate to the GSM algorithm that an external update operation has been requested. Calling this function is not an indication that it is safe to proceed with the update. The update should be attempted only when the GSM transitions into the `UPDATE_PREPARE` state.

5.2.4 Heartbeat Communication

Heartbeat message are transmitted to all remote GSM controllers using UDP broadcast packets. A well known UDP port is monitored by all GSM controllers and any heartbeat message broadcast on that port will be seen. GSM controllers are responsible for broadcasting heartbeat messages to remote GSM controllers and receiving heartbeat messages from remote GSM controllers on the well known UDP port. Below, we describe the circumstances that cause a heartbeat message to be broadcast to all remote GSM controllers.

On startup, the GSM controller establishes an alarm that will trigger at regular intervals. In our prototype system, the alarm was set to trigger every 200ms. When an alarm triggers, the signal handler sets a flag and the appropriate bit in the `fdset` structure corresponding to the UDP broadcast socket. At some point in the infinite loop, the `select` call will return, indicating that data can be written to the UDP broadcast socket. At this point, the GSM controller calls the `gsm_tick` function to generate a new heartbeat message. This heartbeat message is then sent to all remote GSM controllers by writing the heartbeat message to the UDP broadcast socket. Once the heartbeat message has been completely written to the UDP broadcast socket, the flag is cleared. The next time the alarm triggers, this process is repeated.

5.2.5 Querying the GSM State

At times, it is necessary for external components to query the state of the GSM. An example is the IO Proxy, who, without up to date information regarding the state of the distributed group, would be unable to provide the reliable inter-node communication required. The GSM controller must allow multiple query clients.

When an external component wishes to query the state of the GSM, the component establishes a TCP connection to the GSM controller using the well known TCP port. After the connection has been accepted by the GSM controller, the component now has five commands it can use to query the state of the GSM. Table 5.3 shows the available commands.

The API is straightforward. The component has the choice of format output for the heartbeat message. By default, heartbeat messages are sent in text format. This format option is used primarily for debugging as one can easily connect to the GSM controller

`@raw`	Send heartbeat messages in raw byte format
`@text`	Send heartbeat messages in text format
`@next`	Send the next heartbeat message only
`@start`	Start sending every heartbeat message
`@stop`	Stop sending heartbeat messages

Table 5.3: GSM Query API

using `telnet` and query the state of the GSM. The only other decision for the component is whether it is interested in the current state only, or, if it is interested in always knowing the most up-to-date state of the GSM. The IO Proxy is an example of a component that must always know the most up-to-date state of the GSM and as a result, the IO Proxy uses the `@start` command.

The final detail with regards to querying the GSM state is how the heartbeat messages are sent back to the component. Immediately after broadcasting the most recent heartbeat message to all remote GSM controllers, the local GSM controller copies the heartbeat message to the buffer associated with each query component. The associated bits are set in the `fdset` structure and over the course of one or more `select` calls, all components querying the state of the GSM will be sent a copy of the last generated heartbeat message. When a heartbeat message has been entirely sent, the appropriate bit in the `fdset` structures are cleared and no further heartbeat messages will be sent (unless the query component has previously asked for all heartbeat messages to be sent).

5.2.6 Updating the GSM State

We have already seen that, at times, it is necessary for the GSM to wait for external input before transitioning between states. Only one external update client is allowed to provide input to the GSM. After accepting an incoming connection from an update client, the GSM controller monitors the update client for activity in addition to the regular set of communication channels it monitors. The three commands that can be sent to the GSM controller are listed in table 5.4. When the GSM controller receives one of these commands, it calls the appropriate function in the GSM API previously described.

`UPDATE`	Request a global update operation to take place
`ABORT`	Abort the current distributed transaction
`COMMIT`	Commit the current distributed transaction

Table 5.4: GSM Update API

5.3 Four11 Server

5.3.1 Overview

The sole task of the Four11 server is to act as a central registry where components can publish port numbers for publicly available services. This is the same task provided by the RCP portmap service [16]. Any component wishing to use a service of another component must first query the local Four11 server of the component offering the service. No persistent state is stored by the Four11 server. If a node fails, all information within the Four11 server is lost. This is the desired behavior. Figure 5.4 shows the two possible situation. In the figure, components $C1$ and $C3$ need a service offered by component $C2$.

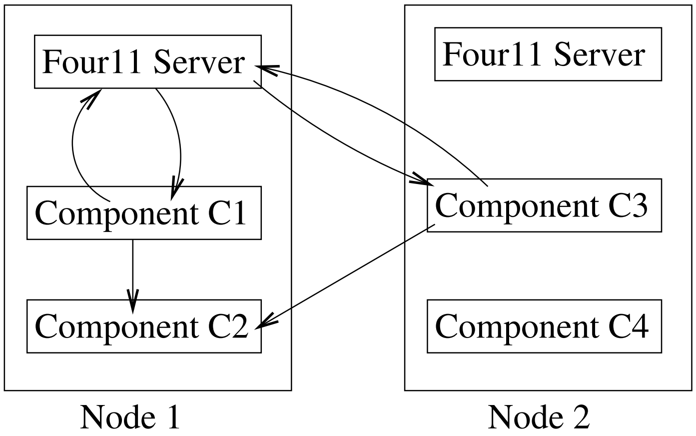

Figure 5.4: Four11 Server Interaction

The Four11 server is a multi-threaded process. The reason for using multiple threads instead of a single thread, is simplicity. Since each client has its own thread of execution, non-blocking IO, and its added complexity, is not required. The Four11 server consists of an infinite loop, within which new client connections are accepted. When a connection is accepted, a new thread is created to handle the client's requests. The client thread is also structured around an infinite loop in which client commands are read, processed, and a response is written back to the client. The loop repeats itself until the client closes the connection, at which point, the thread is destroyed.

Two data structures are needed for the Four11 server to perform its duties. A *registered* list and a *waiting* list. The registered list contains all the services that have been registered so far, while the waiting list contains a list of services that have been requested but have not yet been registered. The registration consists of a unique identifier port consists of a unique identifier and a port number. Multiple registrations by a single component are allowed provided each port is assigned to a unique identifier. Name conflicts can be avoided by using a simple naming convention. When a client performs a query for a port number, the client supplies the unique identifier of the service whose port number is desired. These identifiers must therefore be known *a priori*. When a component registers a port number with a unique identifier, the identifier and port are added to the registered list. The waiting list is useful in helping avoid race conditions. Presumably, a component would not ask for the port number of a service if that service did not exist, or would not exist in the very near future. If the Four11 server sees a request for a service identifier that has not yet been registered, it is reasonable to expect that service to be registered shortly. The waiting list allows the Four11 server to keep track of what services have been requested in hopes of being able to successfully answer those queries momentarily.

5.3.2 Communication Channels

The Four11 server listens for incoming TCP connection on a well known port. This is the only point of contact for any component to communicate with the Four11 Server. The Four11 server is a self-contained component and does not required the services of any other RDSS component.

5.3.3 Client Commands

There are three main types of client commands. A client can *query* for a service port, it can *register* a service port and finally it can *release* a service port. Within each of those command types, there are further specifications. Table 5.5 shows all commands available to a client.

`@register <id> <port>`
`@release <id>`
`@release_all [prefix]`
`@query <id> <timeout>`
`@query_all [prefix]`

Table 5.5: Four11 Client Commands

Registering Service

To register a service and port tuple with the Four11 server, the client sends the `@register <id> <port>` command. Upon receiving this command, the Four11 server attempts to enter the tuple in the registered list. Since service ids are required to be unique, if a tuple already exists in the registered list with the same service id as the one given by the client, the new tuple is not added and an error message is sent to the client.

If no previously existing tuple is found in the registration list with a service id of *id*, the tuple $< id, port >$ is added to the registration list. After making the entry, the Four11 server then scans the waiting list for any requests for the service id *id*. Any thread found waiting in the waiting list for the service id *id* is notified that the service has now been registered. Upon waking up, each of the previously sleeping threads will return the result to their respective clients.

Releasing Services

There are two variations of the command used by clients to release any tuples previously registered. The @release <id> and the @release_all [prefix] command. The @release <id> command instructs the Four11 server to search the registration list and remove the tuple whose service id is id. If the tuple is found, it is removed and a successful response is sent back to the client. If the tuple can not be found, an appropriate error is returned instead.

The @release_all [prefix] command is a convenience method to allow a component to release several services with a single command. This command causes the Four11 server to scan the registration list and any tuple found whose service id begins with prefix is removed from the registration list. This command always returns a successful response to the client. This command is extremely useful when a component chooses to prepend a common prefix to every service id it registers. Care must be taken to ensure that no other component uses the same prefix or that components services will also be released.

Querying Services

To query a service, a client sends the command @query <id> <timeout> command. id is the service id whose port is to be returned. The timeout value is an integer that specifies how long the client is willing to wait in the event of an unsuccessful search.

The first step performed by the Four11 server is to scan the registered list. If a tuple exists whose service id is id, the port number from the tuple is returned to the client. If no entry is found in the registration list, the four11 server now looks at the timeout value specified. The timeout value specifies how many seconds the four11 server should wait for a registration for the service id id to occur. A value of 0 means that the client thread should not wait and should return an error response immediately after an unsuccessful search. A positive timeout value means that the client thread should wait that many seconds before returning an error response to the client. A negative timeout value means the client thread should not return a response to the client until a registration for the service id id occurs (infinite timeout value).

Timeouts are implemented using condition variables; there is one condition variable for each waiting thread. When a sleeping thread wakes up, there are two possible reasons why the thread was woken up. One possibility is that a registration occurred and the condition variable was explicitly signaled by another client thread. In this case, an tuple will now exist in the registration list for the service id *id*. The woken thread should rescan the list and return, to th client, the port number associated with the service id *id*. Alternatively, the client thread woke up because it timed out while waiting on the condition variable. In this case, the service id *id* was not registered and the four11 server should return an unsuccessful search response to the client.

The last command `@query_all [prefix]` is the complimentary command to the `@release_all [prefix]` command. By returning all services registered with a command prefix, a client can learn of all services available by a component. This command is useful only when a component uses a common prefix to register its services. In our prototype system, the `@query_all [prefix]` command is used during a node removal operation. Any services associated with the removed node must be released from the four11 server. The `@query_all [prefix]` command is a convenient command to accomplish this task. No timeout values are permitted with this command.

5.4 Node Controller

5.4.1 Overview

In our design discussion, we mentioned that the Node Controller is responsible for managing the actions of all components during a distributed group operation. While the actions required of the Node Controller are many, individualy they are simple.

5.4.2 Communication Channels

For the most part, when the Node Controller needs to communicate with other components, the Node Controller is the component that initiates contact. At startup, the Node Controller establishes two connections to the local GSM controller. The first connection is a query connection used to monitor the state of the GSM. The node controller must respond to transitions into a `PREPARE` by the GSM and it is through this query connection that the Node controller will learn of such a transition. Since the Node controller manages all activities of a node when performing any group operation, the Node controller establishes a second connection to the GSM controller as the updater (there is only one since the GSM controller allows only one update connection). Using the update connection, the Node controller can request external updates to take place as well as provide input regarding the success/failure of the operation currently being attempted.

The only one reason for an external component to connect to the Node controller is to request an update to occur in the distributed system. Components wishing to perform an external update must first connect to the Node Controller using a well known TCP port. In addition to monitoring the GSM controller for heartbeat messages, the node controller also watches for incoming update connections.

The last aspect of IO is in node synchronization. Recall that node synchronization is implemented using one central node controller. A potential race condition exists between the central node controller accepting incoming connections and all remote node controller establishing said connections. We avoid the race condition issue by having all node controllers be prepared to accept incoming connections in advance. At startup, each Node Controller sets up a socket to listen for incoming TCP connections.

5.4.3 Resource Management

The Node controller is responsible for ensuring all necessary application instances are running on the local node. When adding a new node to the distributed group, the Node Controller must create a new instance of the application and a new instance of the application proxy. The new application and application proxy instances created when performing an node addition operation become the foreign secondary application and foreign secondary application proxy instances associated with the node being added. Similarly, during a node removal, the foreign secondary application and application proxy instances associated with the node being removed must be destroyed upon successful completion of the operation.

Both the application and the application proxy require a minimum set of resources in order to function properly. Additionally, configuration files or other setup files might also be required in order to successfully create new instances of the application and the application proxy. It is important for the node controller to know these resources in advance to allow it to create any new instances and also to recover resources when an application or application proxy instance is destroyed.

Some permanent storage must be available for both the application and the application proxy.The Node Controller must know where the working space exists and how much there is to use. The amount of storage needed for the application is dependent on the application itself. Sufficient storage is needed to store all work to be performed for an operation. Additional storage is needed to build a new master copy of the document collection during the `PREPARE` stage of the operation. After the prepare stage, there are two copies of the document collection representing the state before the operation and after the operation. Typically, our prototype system required roughly three times the storage requirements of a single copy of the document collection. The storage requirements of the application proxy are discussed in section 5.6.

In our prototype system, the only resource required is a working directory. For simplicity, our prototype system assumes that sufficient storage capacity exists in the working directories. The ramafications of removing this assumption would be that the application would have to verify it would not exceeded its resource limits if it proceeded with the operation. The Node Controller was provided with a configuration file defining a set of working directories that could be used by various application and application proxy instances. The Node Controller used this list when assigning working directories to any new application or application proxy instances. The node controller also maintained, on permanent storage, a list of resource that had already been assigned to currently existing application and application proxy instances. The list of currently used resources was stored on permanent storage to allow the Node Controller to restart all application and application proxy instances in the event of a node failure.

5.4.4 Component Management

One of the guiding assumptions during our design and implementation was that individualy, the various components, specifically the application and application proxy instances, have no concept that they belong to a larger distributed system. The node controller is primarily responsible for managing the distributed transaction and for informing components when it is their turn to do some work. Two options were available for managing the transaction. The first option was to have each Node controller manage the transaction for the primary application instance and all remote secondary application instances. The second option was for the Node Controller to manage the primary application instance and all foreign secondary application instances. In our implementation, we chose to have the node controller use the latter option. The reason for this decision was to avoid complications when committing or aborting the transaction. The problem that arose was that after a global decision to commit a transaction has been made, there could be communication problems between the node controller and the remote secondary applications. By choosing to have the node controller control the transaction progress of components on the local node only, this problem does not exist.

5.4.5 Node Synchronization

Before any synchronization can occur, the master node controller must first accept connections from each of the slave node controllers. The concept of master and slave node controllers was introduced in chapter 3, section 3.3.6. This is the first task performed when performing any group operation. When establishing the synchronization connection, each node controller inspects the heartbeat message which caused the operation to occur. If the local node is listed as the king node controller in the heartbeat message, the node controller knows that it is the master node controller and it should accept incoming connections from all remote node controllers. If the local node is not the king node, the local node controller attempts to establish a connection to the king node controller using the well known synchronization TCP port.

Normally, the king node controller will wait for all remote node controllers to establish a connection. However, this solution is not resilient against failures by a remote node before that node's node controller successfully establishes a synchronization connection. The problem is that if a node fails before a synchronization connection is established, the king node controller will never learn that a node has failed. The solution used in our prototype implementation was to have to king node controller wait for all remote node controller to connect up to a maximum of 30 seconds. Since the first task performed by any node controller is to establish a synchronization connection, it is reasonable to expect the connection to occur very quickly. A better solution would be for the king node controller to monitor the local GSM controller for more recent heartbeat messages. Doing so would allow the Node controller to better understand the state of all nodes in the system and would avoid the necessity of setting an arbitrarily chosen timeout value.

Once the establishment of all synchronization connections has been completed, synchronizing is a simple task. The king node controller reads the votes from all remote node controllers, computes a final outcome and writes the outcome back to each remote node controller. If an error occurs when performing an operation, a node controller immediately aborts the operation, closes all connections and informs its local GSM controller. If the king node controller closes its connections, each of the slave node controllers will notice the closed connection at the next synchronization point. If a slave node controller closes its connections, the king node controller will notice at the next synchronization point. Similarly, if a a node failure occurs, the IO Proxies will shut down the appropriate connections and again, all node controllers will notice that one or more synchronization connections have been closed. It is assumed that a premature closing of a connection is an indication of an error and that the current operation should be aborted. We assume for the duration of this section that each synchronization point is successful and the current operation will proceed beyond the synchronization point. If a synchronization point is not successful, the current operation is immediately aborted by all node controllers.

5.4.6 Node Controller State

While we tried to make the node controller as stateless as possible, there are a few pieces of information that must be maintained on permanent storage. We have already discussed the need for the Node controller to store on permanent storage the list of resources available and the list of resources currently assigned to various RDSS components. The Node controller must also store the most recent snapshot of the distributed system, according to the local GSM. The snapshot is simply a heartbeat message containing the most up-to-date membership of the distributed group. It is this information that is used during a restart to decide what, if any, application instances should be started to restore the node to the state that existed before the failure occurred.

The Node controller must also maintain some state regarding the status of the current distributed transaction. For the purposes of a restart, the node controller must know what it was last doing before the failure occurred in order to properly restart the node. If the node was in the process of performing a transaction, but no decision had been made regarding the committal or abort of the operation, the node controller can simply abort the transaction because the transaction would have been aborted globally since the node had failed. However, if the node controller was in the process of informing all components to either abort/commit a transaction and a failure occurs, the node controller must know that it was in the processing of notifying the components since all components will, upon restart, wait for input from the node controller regarding the commit decision of the last operation being attempted.

5.4.7 Beginning an Operation

Between the various operations to be performed, there are several common actions that take place at the beginning of the operation. The establishment of the node synchronization connections is the first step. The next step is to establish a connection to the local Four11 server. This connection is necessary in order to learn the port numbers being used by the many services of all local components. Finally, a connection is established to the local distribution server. At some point during the operation, the distribution may have to do some work. The Node controller will inform the distribution server using this connection.

5.4.8 Node Addition

When the GSM decides that a node is to be added to the distributed membership, the GSM state will change to `HB_ADD_PREPARE`. The node controller will notice the state change and will proceed to guide the node through all necessary steps. For clarity, let us say that the current node addition operation is increasing the distributed membership from n nodes to $n+1$ nodes. The steps performed by the node controller are broken down below. Figure 5.5 gives an overview of all steps required.

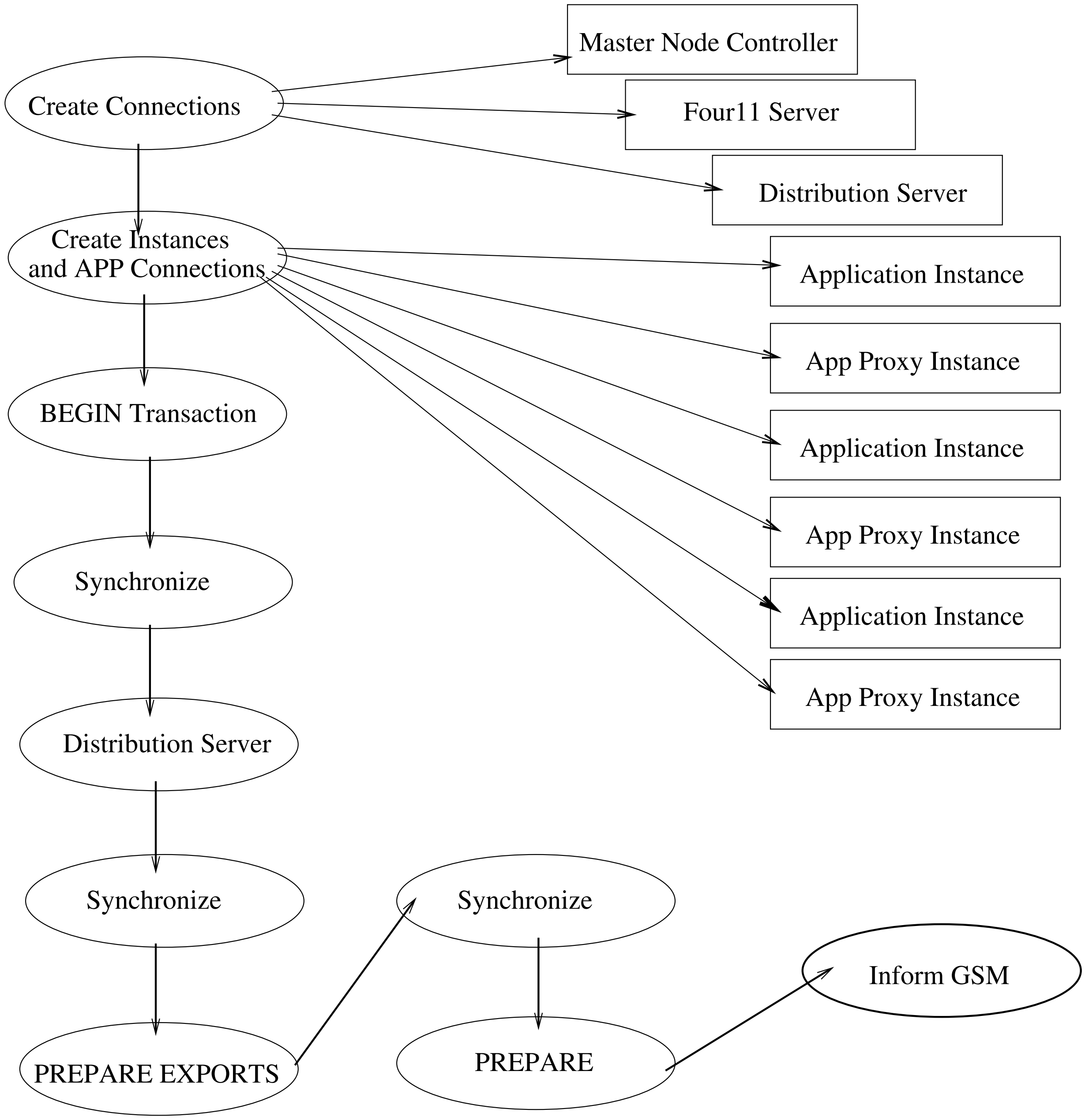

Figure 5.5: Add Operation: Node Controller Steps

Step 1: Establish Connections

Establish the connections to the four11 server, the king node controller and the distribution server.

Step 2: Create Application/Application Proxy instances

If the node controller is running on a previously existing node, the node controller must instantiate one application instance and one application proxy instance. These instances become the foreign application instances of the new node. If the node controller is on the node being added, the node controller must instantiate one primary application instance and one primary application proxy instance. Additionally, it must also instantiate n foreign secondary application and application proxy instances.

Step 3: Begin Transaction

The Node Controller must connect to each application and application proxy instance on the local node. The connection is established with the update service of the components. This connection is used for guiding each instance through the distributed transaction only. At this point, we refer back to the timeout service of the four11 server. The race condition at play is that the application and application proxy instances must register their update port with the four11 server before the node controller can query for the port. In our implementation, the node controller specifies a timeout period of 30 seconds. This should be more than ample time for the potentially $2n$ application and application proxy instances to be created and have each process register their ports with the four11 server. If an application or application proxy instance is unable to register its update port within 30 seconds, the node controller assumes that the creation of the application and application proxy instances failed. The last step in preparing the application instances for the upcoming operation is to notify them to start a transaction. The command `APP_BEGIN` is sent to each application and application proxy instance.

Step 4: Synchronize

Synchronization must occur to ensure all application and application proxy instances in the distributed group have been told to begin a transaction. No further progress can be made until such time.

Step 5: Distribution Server

The next step is to instruct the distribution server to take whatever steps are necessary according to the operation being performed. The node controller is unaware of what the distribution server does. The command `DIST_GSM_UPDATE` is written to the distribution server in addition to the heartbeat message that caused the update to occur. The heartbeat message is needed by the distribution server to decide what actions should take place. At this point, the node controller waits for the distribution server to indicate the success or failure of the work it performed. After reading the result from the distribution server, another synchronization point occurs.

Step 6: Synchronize

Synchronization is necessary to ensure all distribution servers have finished their work before further progress can occur.

Step 7: Prepare Exporters

The command APP_PREPARE_EXPORT is written out to all application instances and application proxy instances. When all instances have reported back that all exporters are ready, the node controller proceeds to the next step.

Step 8: Synchronize

Before the application and application proxies can prepare their work, all exporters in the distributed group must be ready. A synchronization point is used to guarantee this requirement.

Step 9: Prepare Work

The command APP_PREPARE is written out to all application and application proxy instances. At this point, each application and application proxy instance performs all actions that were dictated to it by the distribution servers. The node controller waits to hear from all instances and when all instances have reported back, the final step occurs.

Step 10: Report to GSM

The last step in preparing an operation is to report back to the GSM controller the success or failure of the operation. This information will be used by the GSM algorithm to decide the final outcome of the distributed transaction.

5.4.9 Node Removal

From the perspective of the node controller, removing a node from the distributed group is similar to adding a node. The event that causes the node controller to remove a node is a transition by the GSM into the HB_REM_PREPARE state.

The only difference between adding a node and removing a node is that the node controller does not have to create any new instances when removing a node. Additionally, at the completion of a successful node removal operation, the node controller must destroy the foreign application and application proxy instances associated with the node that was removed. Figure 5.6 gives an overview of all steps required.

5.4.10 External Update

The set of operations required for performing an external update are similar to those for adding and removing a node. The state transition that causes a node controller to perform an update operation is the `HB_UPD_PREPARE` state. The main difference is that the node controller does not give any commands to its local distribution server. Since an external client is performing the update, it is up to the external client to give instructions to any necessary distribution servers within the distributed group.

For the purposes of our discussion, we differentiate between the *initiating* node controller and the *responding* node controller. The node controller who accepted the external update client and told its GSM to begin an update operation is referred to as the initiating node controller while all other node controllers are the responding node controllers. We examine the steps from both perspectives. We assume the basic 10 steps as outlined in the node addition algorithm.

Responding Node Controller

A responding node controller follows the same basic steps as those used when adding or removing a node. The difference is that since the distribution server is not involved, step 5 is not needed. A modified version of the steps required for a responding node controller are outlined in figure 5.7.

Initiating Node Controller

The initiating node controller becomes involved in the update operation well before the `HB_UPD_PREPARE` state transition occurs. All node controllers listen for incoming connections requesting a global update operation to take place. For a particular update operation, the initiating node controller happens to be the node controller that was contacted by the external client. After accepting the connection, the node controller uses its GSM update connection to request a global update. At this point, the node controller does nothing until the `HB_UPD_PREPARE` state transition is seen. During this time, the external update client is forced to wait.

Once the `HB_UPD_PREPARE` state is seen, the node controller follows the same set of steps. Where the node controller would normally inform its local distribution server to do its work, instead, the node controller informs the external update client to do its work. The node controller waits for the external update client to write back to the node controller

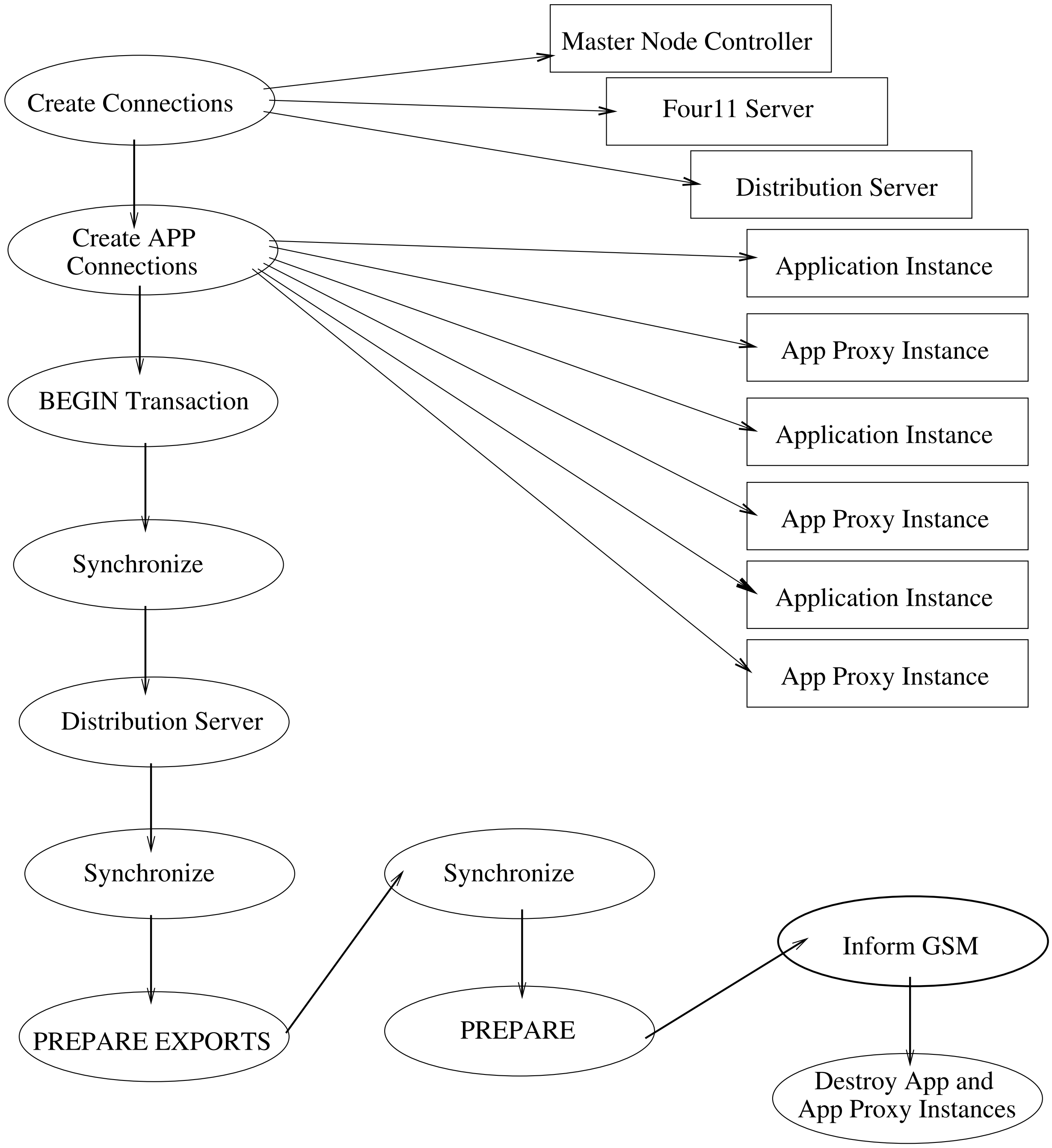

Figure 5.6: Remove Operation: Node Controller Steps

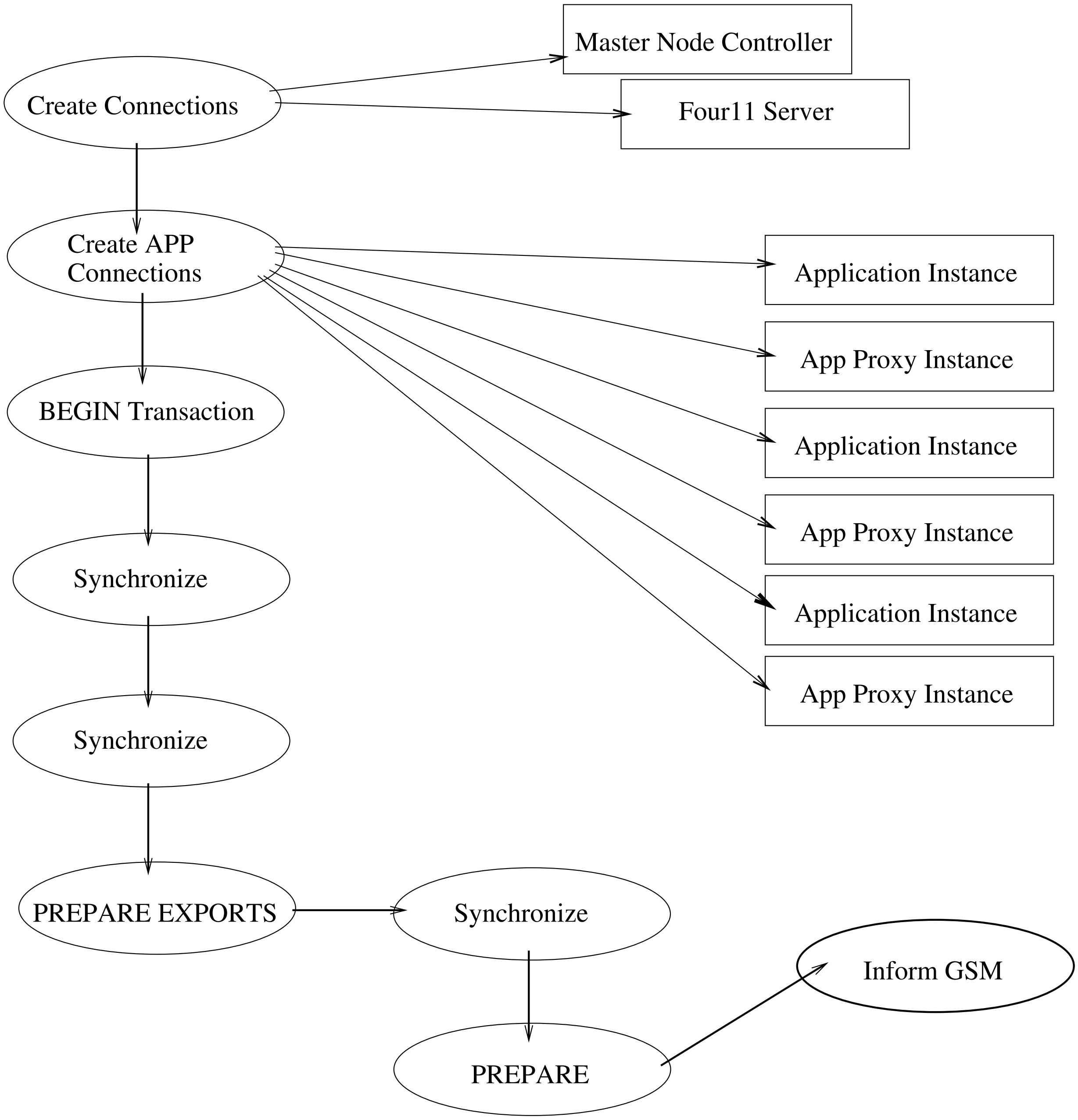

Figure 5.7: Update Operation: Responding Node Controller Steps

informing it that all work has been handed out and the transaction can proceed. A modified version of the step required for an initiating node controller are outline in figure 5.8.

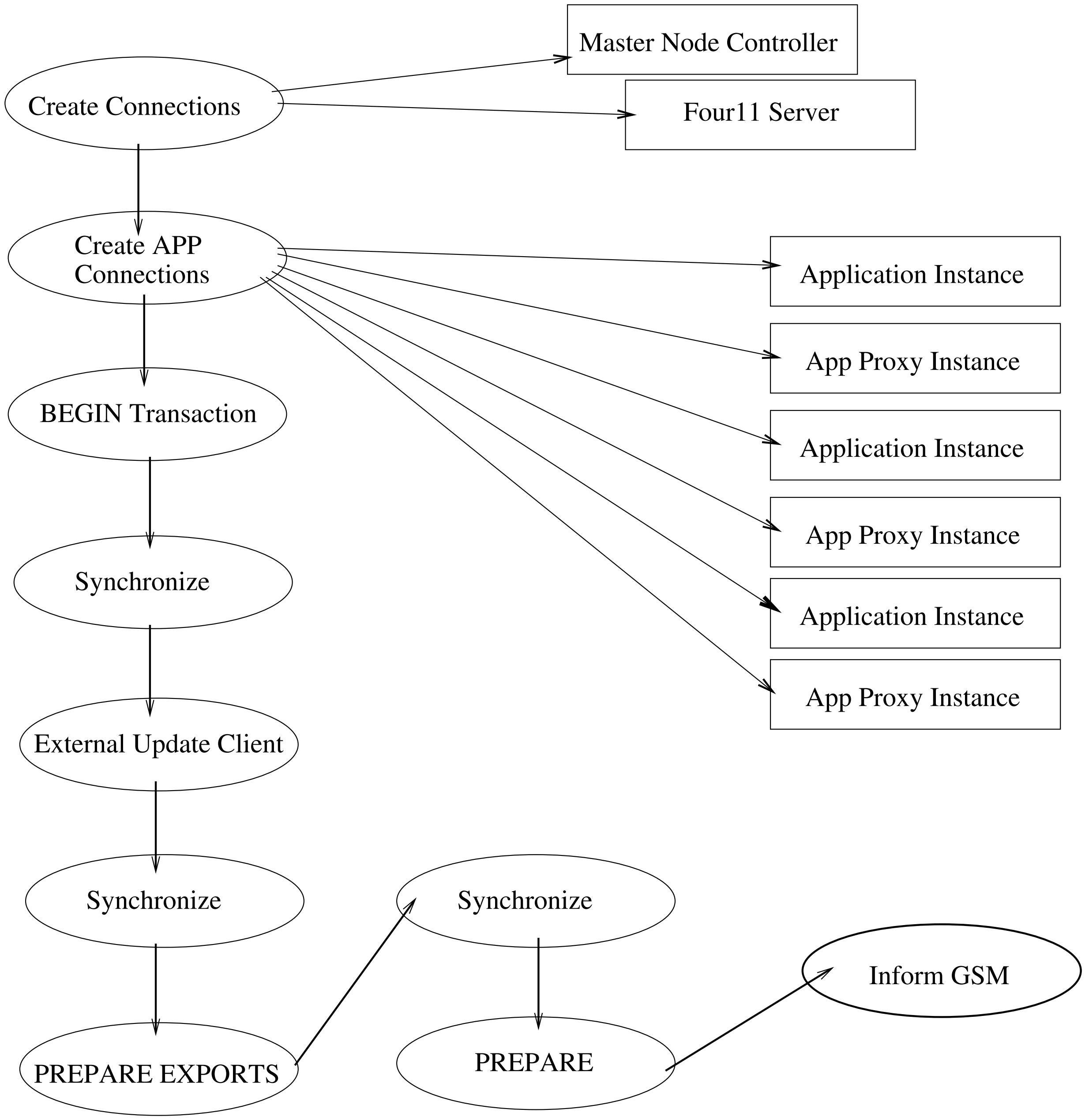

Figure 5.8: Update Operation: Initiating Node Controller Steps

5.4.11 Transaction Completion

When the node controller has told its local GSM controller about its ability to commit or abort the transaction, the node controller takes no further action until the outcome of the distributed transaction is known. Each heartbeat message contains a count for of the number of attempted operations and a count of the number of committed operations. The node controller remembers the heartbeat message that caused the distributed transaction, and after comparing with more recent heartbeat messages, the node controller can learn whether the distributed transaction was committed or aborted.

Suppose a heartbeat message is received in which a distributed transaction is to take place. Let us assume that the number of attempted transactions was a while the number of committed transactions was c. When a decision as been made, the GSM will increment the attempted operation count, and if successful, will also increment the committed count. If a future heartbeat message has attempted and committed count numbers of $a + 1$ and c respectively, we know that the distributed transaction was aborted. A committed count number of $c + 1$ indicates that the transaction was committed. Under no failures, it is impossible for the attempted count number to be anything but $a + 1$. The only situation where the attempted count can be higher that $a + 1$ is if the node fails and that node is then removed by the remaining members of the distributed group.

Some state is stored to permanent storage by the node controller for the purposes of transaction management. Any heartbeat message that causes an operation to take place is stored. Additionally, the node controller stores some information regarding the progress of the current transaction. These pieces of information are necessary for the node controller to recover after a node failure. When committing or aborting a transaction, the final step is to store the most recent heartbeat message to permanent disk as the heartbeat message is currently the most accurate description of the distributed group

5.4.12 Transaction Commit

The first step in committing a transaction is to inform all application and application proxy instances that the transaction is to be committed. The subsequent steps depend on the operation being performed. If the operation was to add a new node to the distributed group, the resources that were assigned to the new node must be recorded as used by the foreign secondary application and application proxy instances. If the operation was to remove a node, the foreign secondary application and application proxy instances must be destroyed and their resources reclaimed. Again, the new state of the resource usage must be stored to permanent disk. An external update operation requires no additional work by the responding node controllers. The initiating node controller must respond to the external update client indicating whether the operation was successful.

5.4.13 Transaction Abort

Much like committing a transaction, the node controller's first step is to inform all application and application proxy instances that the operation is to be aborted. Following that, if the operation was a node addition, the node controller must destroy the foreign secondary application and application proxy instances that were created for the new node. No addition steps are needed to abort the removal of a node. An external update requires only the initiating node controller to inform the external client that the operation was unsuccessful.

5.5 Distribution Server

5.5.1 Overview

The distribution server is responsible for implementing the RDDS throughout the distributed system. The distribution server is a multi-threaded process.

Each connection to the distribution server results in a separate process to handle the update client. There are two types of clients. The client is either an external update client wishing to perform an update operation, or, the client is the local node controller. After accepting a connection from a client, the client must tell the distribution server what type of update to perform.

5.5.2 Communication Channels

The Distribution Server listens for incoming client connections on a well known TCP port. This is the only point of contact for any component wishing to make updates to the global document collection.

5.5.3 Document Collection Updates

After establishing with the distribution server and sending the `DIST_EXT_UPDATE` command, the external client now has several options in terms of making document collection updates. There are six commands that can be used by an external client to make updates to either the global document collection, or a particular node's primary document collection. The commands are given in table 5.6.

All transactions consisting of document collection updates must begin with a `DIST_EXT_BEGIN` command and end with a `DIST_EXIT_END` command. It possible for several external clients to make document collection updates as part of a transaction, but exactly one of those clients must begin and terminate the transaction with these two commands. If there are

DIST_EXT_BEGIN
DIST_EXT_END
DIST_GLOBAL_ADD
DIST_PRIMARY_ADD
DIST_GLOBAL_DELETE
DIST_PRIMARY_DELETE

Table 5.6: Document Collection Update Commands

multiple external clients for a single transaction, it is up to the clients to coordinate amongst themselves so that the protocol is followed correctly. Between the two commands marking the beginning and end of a transaction, any sequence of the remaining four commands can be sent to the distribution server. Below we discuss each of these six commands and what the distribution server does when the commands are given.

Begin Update Operation

A global add operation is initiated by sending the `DIST_GLOBAL_ADD`. Upon receiving this command, the distribution server proceeds to contact the node controller on the well known port to initiate a global update transaction. When the node controller has received confirmation that the distributed transaction can proceed, the node controller will respond to the distribution server indicating that it may proceed. The distribution, in turn, notifies the external update client that it may proceed with its update.

End Update Operation

The command `DIST_EXT_END` is sent to the distribution server once all update operations to be performed have been sent to the distribution server. Upon receiving this command, the distribution server informs the node controller that all work to be performed as part of the update operation has been given to the appropriate application and application proxy instances. In a transaction with many external clients handing out work as part of the transaction, it is a requirement that the same client that initiated the distributed transaction by sending the `DIST_EXT_BEGIN` must also terminate the update by sending the `DISt_EXT_END` command. The reason for this is that only one distribution has a connection to its local node controller. It is this same distribution server that must report back to its local node controller regarding the success or failure of distributing the work for the update operation.

Primary Document Collection Addition

The command DIST_PRIMARY_ADD is sent to initiate this operation. This operation is used to add documents to the primary application instance and its corresponding remote secondary application instances. As such, only those application instances are involved. The necessary connections to these application instances are created. The APP_ADD command is sent all application instances to prepare them to accept some incoming documents that are to be added to their document collection.

Each document to be added to the primary document collection is read from the client and sent to the primary application as well as one of the secondary application instances. The document additions are distributed to the remote secondary application instances in a round robin manner. As discussed in chapter 3, section 3.4.3, doing so guarantees axioms 3 and 4 of the RDDS requirements. By distributing the documents in a round robin manner, the balanced nature of the distribution across all remote secondary application instances is not affected. If the distribution was even to begin with, the distribution is still even. No effort is made to improve a balanced distribution, but at the same time, an effort is made to ensure a balanced distribution is maintained.

Primary Document Collection Delete

The command DIST_PRIMARY_DELETE is sent to initiate this operation. This operation is intended to delete a set of documents from the primary application instance and its corresponding remote secondary application instances. Consequently, this operation involves only the primary application instance of the local node and all remote secondary application instances. After establishing the necessary application connections, the APP_DELETE command is sent to each application instance informing it to accept a list of incoming document that should be deleted. An additional connection to each of the remote secondary application instances is also created. The APP_LISTING command is sent these extra connections. It is through this supplementary connection that the distribution server learns about what documents are stored on each remote secondary application instance. The connection used for sending updates we call the *update* connection and the connection used for retrieving a list of stored documents the *listing* connection.

Once all commands have been sent to the appropriate connections, the distribution begins reading the unique document identifiers corresponding to the documents to be deleted. As the document ids are read, a co-sequential scan is performed on all listing connections. If a document is found to exist in one of the secondary application instances, the document is then removed from both the primary application instance and the remote secondary application in which it was found. Any documents that are to be deleted but are not found in any of the listing connections is simply discarded. Once all documents to be deleted have been read and evaluated, all connections to the application instances are closed. Referring back to the RDDS definition, axiom 4 is unaffected by our operations since we are not adding any documents. Axiom 3 is maintained because each document removed from the primary application instance is also removed from the appropriate remote secondary application instance. No claim can be made regarding the evenness of the distribution after the operation is complete.

Global Document Add

This command is initiated by sending the `DIST_GLOBAL_ADD` command. This command is used to add a set of documents to the global document collection, and not just that of a specific primary application. Adding documents to the global document collection can be viewed as a collection of updates to each of the primary document collections. This is exactly how we implement global document updates.

Recall that each external update client is given its own thread of execution. Let us consider each thread of execution as an instance of the distribution server. This is not to be confused with the instances of the application instances which are in fact separate entities. In this manner, the instance of the distribution server that exists for the client sending the `DIST_GLOBAL_ADD` command is called the *global* distribution server. The first step of the primary instance is to create an update connection to each of the distribution servers, including itself (recursion). Each of these new connections are referred to as *local* distribution server. Figure 3.11 shows the primary instance by itself with the 4 secondary instances.

The global distribution server now sends the `DIST_PRIMARY_ADD` command to each of the local distribution servers. Next, the global distribution server reads the documents to be added from the external client and distributes them to the local distribution servers in a round robin manner. When all documents to be added have been read, the global distribution server closes all connection to the local distribution servers and proceeds to wait for new instructions from the external client. The action of closing the connections causes the primary distribution servers to exit terminate their respective threads. The distribution servers on those nodes still exist, but there is one less thread of execution.

Referring back to chapter 3, section 3.4.3, distributing the documents in a round robin manner guarantees compliance with axioms 1 and 2 of the RDDS defintion for the newly added documents. Again, no effort is made to guarantee a balanced distribution, however, the actions taken will not destroy a balanced distribution if one already existed.

Global Document Collection Delete

This command is used for removing documents from the global document collection. It is the result of the DIST_GLOBAL_DELETE command. Since the update applies to all primary application instances and their respective remote secondary application instances, the same used for global document collection additions is used here.

We begin our discussion by assuming that the global distribution server has already established all the connections to the local distribution servers and the DIST_PRIMARY_DELETE command has been sent. When reading document idenfifiers to be deleted, the global distribution server does not need to worry about what documents are stored each of the primary document collection. Instead, the global distribution server naively sends each document id to all local distribution servers. It is the local distribution servers that will discard document deletions if they do not apply to their primary document collection.

Axiom 2 of the RDDS is maintained because we are not adding any documents to any of the primary document collections. Axiom 1 is maintained because each document deleted from the global document collection is deleted by exactly one of the primary document collections. In the event that a document that does not currently exist in the global document collection is asked to be deleted, our steps are still correct because we can not delete a document that we do not currently have.

5.5.4 GSM Updates

Node Addition

Section 3.4.1, outlines the three steps necessary to redistribute the global document collection over the new distributed group membership. Let N_r denote the node being added. Let N_c be one of the previously existing nodes that must perform the actions outlined below as part of a node addition operation.

We already saw that to ensure an even distribution, we select the subsets r_{ic} individually from each of the previously existing remote application instances instead of choosing r_c from the primary application instance. The subsets P_{ic} are chosen in a similar manner. The actual implementation is straightforward. For each of the previously existing remote application instances, the distribution server opens seven connections. These connections are used to performed the necessary subset selection and document transfers. When all transfers have been performed, the connections are closed. The distribution server performs the same operations on each of the previously existing remote application instances. The seven connections are outlined in table 5.7.

Connection C_1 is used to acquire a listing of the documents stored in S_{ic}. The connections C_2 and C_3 are used to re-balance the primary document collection over n remote

Connection Name	Connected to	Command
C_1	S_{ic}	`APP_LISTING`
C_2	P_c	`APP_EXPORT_DELETE`
C_3	P_a	`APP_IMPORT`
C_4	S_{ic}	`APP_EXPORT_DELETE`
C_5	S_{ac}	`APP_IMPORT`
C_6	S_{ic}	`APP_EXPORT_DELETE`
C_7	S_{ca}	`APP_IMPORT`

Table 5.7: Node Addition Connections

application instance from the current $n-1$ remote application instances. These two connections are used to relocate the subset r_{ic}. The connections C_4 and C_5 are used to relocate the subset p_{ic} from P_c to P_a. Finally, the connections C_6 and C_7 are used to relocate the subset p_{ic} from S_{ic} to S_{ca}.

A single sequential scan is made of the document listing coming over the connection C_1. The sequential scan is performed by reading the document identifiers in groups with $n+1$ document identifiers per group. From this group of $n+1$ identifiers, two document identifiers are chosen. We refer to the two document identifiers chosen as d_1 and d_2. Document identifier d_1 is added to the subset r_{ic} and d_2 is added to the subset p_{ic}. The document identifier d_1 must be written to connections C_4 and C_5. The document identifier d_2 is written to connections C_2, C_3, C_6 and C_7.

In section 3.4.2, we explained that the subsets r_{ic} and p_{ic} need to be $1/n+1$ each of the document collection size stored in a remote application instance S_{ic}. Our selection criteria does indeed choose subsets of that size. It follows that our node addition algorithm guarantees an even, valid RDDS distribution if an even distribution exists at the commencement of a node addition operation.

Node Removal

Let N_f denote the node being removed. Let N_c be one of the previously existing nodes that must perform the actions outlined below as part of a node addition operation.

The node removal process is broken down into two distinct parts. The first part deals with step one of the node removal process as described in section 3.4.1. During this step, the documents stored in the remote secondary application instance S_{fc} are redistributed to the remaining remote secondary application instances. One might ask how do we learn the documents that were stored in S_{fc} if the node N_f is no longer available? The solution is to to compare the collection in the primary application instance P_c with the document collections of all of the remaining remote secondary application instances. Any document found in the primary application instance but not found in a remote secondary application instance must have been stored in S_{fc} and hence must be reassigned for replication.

Connection Name	Connected to	Command
C_1	P_c	`APP_LISTING`
C_2	S_{ic}	`APP_LISTING`
C_3	P_c	`APP_EXPORT`
C_4	S_{ic}	`APP_IMPORT`

Table 5.8: Node Removal Connections, part 1

Table 5.8 shows the connections necessary for to perform step one of the node removal operations. Unlike the node addition operation in which a set of connections were open at each node sequentially, the three connections shown here are opened to each of the existing nodes concurrently. Connection C_1 is opened only once while connections C_2, C_3 and C_4 are opened $n-2$ times. Connections C_1 and the $n-2$ C_2 connections are used for performing a co-sequential scan of the document collections stored by the primary application instance and all of the remaining secondary application instances. These connections are used to learn what documents were being replicated in the failed remote secondary application instance S_{fc}. The connections C_3 and C_4 are used for replicating a document in a specific remote application instances. Connections C_3 and C_4 come in pairs and documents are written to these pairs in a round robin manner to ensure a balanced distribution of the document collection stored in S_{fc}. When all documents have been scanned and replicated if necessary, all connections are closed and stage two of the node removal operation proceeds.

Connection Name	Connected to	Command
C_1	S_{cf}	`APP_LISTING`
C_2	P_c	`APP_IMPORT`
C_3	S_{cf}	`APP_EXPORT`
C_4	S_{ci}	`APP_IMPORT`
C_5	S_{cf}	`APP_EXPORT`

Table 5.9: Node Removal Connections, part 1

Table 5.9 shows the connections necessary to solve steps two and three of the node removal requirements. One connection of C_1 is created and $n-2$ connections of C_2, C_3, C_4 and C_5 are created. Connection C_1 is used to scan the list of documents stored in the foreign application instance associated with the failed node. Referring back to chapter 3, section 3.4.1, the document collection stored in this application instance is chosen as the subset P_c from $D(P_f)$. Connections C_2 and C_3 are used to relocate the subset P_c to the primary application instance. This corresponds to step two of the node removal process. Connections C_4 and C_5 are used to re-establish conformance of axiom 3 of the RDDS definition. The set of connections C_2, C_3, C_4, C_5 should be viewed as a set. There are $n-2$ sets of these four connections. As each document is read from C_1, a new set of four connections is chosen in a round robin manner. Each document is then written to each of the four connections. When all documents have been read form C_1, all connections are closed and the document redistribution is complete for the node removal operation.

5.6 Application Proxy

The application proxy is responsible for maintaining a list of the documents currently being stored by the underlying application. All document collection updates are sent to the application proxy and are then forwarded to the underlying application by the application proxy. In order to allow for proper command forwarding, the application proxy must follow the application protocol entirely. The only operation that is not forwarded to the application is the `APP_LISTING` command which is used solely for the purpose of determining which documents are stored in the application.

The application proxy is a multi-threaded component. incoming TCP connection on a single port, but it must allow multiple concurrent connections. When a connection is accepted, a new thread is created to handle the new client connection. When the connection is closed, the thread releases any resources and exits.

5.6.1 Communication Channels

The application proxy listens for incoming TCP connections on a well known port. When a connection is accepted, the application proxy must open a connection to the underlying application on its update port in order to forward any commands.

5.6.2 Overview

Several data structures are need for the application proxy to perform its duties. The main data structure is its list of document headers corresponding to all documents in the document collection of the underlying application. The application proxy implements transactions in a "master/new master" manner. At any point in time, the master copy of the document headers corresponds to the document collection of the underlying application. When performing a transaction, the application proxy builds a new master copy of document headers that will be representative of the document collection after the current transaction is committed. When the application proxy commits a transaction, the old master is deleted and the new master becomes the master copy.

Given that there may be many application proxy instances running on a single node, the application proxy stores all data on permanent storage to reduce the in memory footprint of each instance. When a transaction begins, the application proxy must be ready to accept commands and store any necessary data to be able to apply the changes during transaction preparation. For each command affecting the document collection of the application, the application proxy store the action, as well as the list of documents and any accompanying data to permanent storage. When preparing the transaction, the application will revisit all commands and accompanying data and apply the changes to the mast document list.

There are five types of lists stored by the application proxy while it is gather work to be performed as part of a transaction. Each list corresponds to one of the document collection update commands. Since the application proxy allows multiple clients sending the same commands, it is possible to have multiple copies of one command. These lists are stored in sorted order based on the document identifier. It is imperative that these lists be sorted for the application proxy to be able to efficiently merge the current master list with all the updates. It is not the job of the application proxy to sort the lists as they are sent by update clients. It is the job of the update client to send the lists in sorted order.

For all commands issued to the application proxy (transaction command and document collection commands), a response must be returned to the update client. For all but one of these commands, both the application and the application proxy must perform any necessary work in response to the command. The operation is deemed successful if and only if both the application and the application proxy deem the command to be successful. The application and the application proxy each vote on whether to return `APP_SUCCESS` or `APP_ERROR` to the command issuer. The application passes its vote to the application proxy by returning the appropriate return code to the application proxy. It is the job of the application proxy to select its own vote, read the vote of the application, compute a final response and then transmit that response back to the command issuer.

5.6.3 Command Forwarding

The application proxy forwards commands and any accompanying data to the application on the fly. The list of document headers for which the command applies is read one document header at a time. The application proxy reads one document header and its data (if applicable). The application proxy forwards this document header to the application and then stores the document header to disk. After storing the document header to disk, the application proxy is free to read the next document in the list. The use of blocking IO between the application proxy and the application prevents a fast application proxy from inundating a slow application.

5.6.4 Applying Changes

Changes are applied to the master list by performing a co-sequential scan of the master list and all lists associated with commands that affect the document collection. For clarity, there are four types of lists that must be examined. The *add* lists and the *import* lists contain document headers that must be added to the master list. The *delete* lists and the *export/delete* lists contain document headers that must be removed from the master list. The requirement of sorted lists is needed for the co-sequential scan to be successful.

5.6.5 Storage Requirements

The storage requirements of an individual application proxy instance depends largely on the number of documents being stored in the underlying applicaiton instance. The list of documents stored in the application instance is the main storage requirement of the application proxy. In our prototype system, we assumed that each document header was 128 bytes in size. Thus, to store 1000000 documents in the application instance, the application proxy would need 128000000 bytes of storage just to store the list. It should be noted that in our prototype system, the only useful information was the unique document identifier. We chose to store the document headers as 128 byte header to achieve for realistic performance benchmarks.

When performing a transaction, the application proxy needs space to store the lists of documents that are to be added, deleted, imported and exported from the underlying application. Furthermore, when preparing a transaction, a new list of documents will be build requiring additional space. It is not possible to put a bound on the storage requirements of an application proxy since in theory, it is unbounded. Consider a transaction where an application proxy is told to export its entire collection multiple times. Each export operation would store a list containing all documents stored in the underlying application. The above mentioned situation is likely rare and in our prototype system, the peak storage requirements of an application proxy was three to five times the storage requirements for storing the master list of documents. The three to five factor accounts for two copies of the master lists and some additional space for the lists containing work to be performed.

Chapter 6

Results

6.1 Experiments

There were two types of tests that we ran. The first set of tests were designed to measure the overhead imposed by the RDSS management software only. Under normal operation, we expect the application to dominate the overall time required to perform each type of operation. However, the RDSS does impose some overhead and we sought to quantitatively measure this overhead.

The second test was designed as a a proof of concept. We chose a previously existing IR application for which we had no input in its design or implementation. The goal was to write a layer of software that would make the chosen application implement our application protocol. In theory, if we could make the application support our application protocol, all the benefits of the RDSS would be realized by the application.

6.1.1 Cluster Configuration

Our test system includes a hybrid cluster of off-the-shelve PCs connected via gigabit Ethernet. All workstations ran a version of the Linux operating system. The processing capabilities as well as the storage capacities of the workstations varied. Some of the workstations contained dual pentium 3 processors 1 1GHz with 1GB of ram. The rest of the workstations were dual athlon 1900+ processors with 2 GB of ram.

6.2 RDSS Overhead

To measure the performance impact of the RDSS only, we needed an application that fully implements the application protocol, but would not impose any overhead when the RDSS is redistributing the document collection. For this task, we created an *null* application, which fully implements the application protocol but does not perform any storage or retrieval functions. Documents that are to be added or deleted are simply discarded. When a request arrives for a set of documents to be exported, the null application would set up a port on which the importer could connect. When the importer connected, the null application would immediately close the connection. Similarly, when an import request arrives, the application would connect to the exporter and then close the connection immediately thereafter. Since no data is stored, we expect any overhead due to the null application to be negligeable.

6.2.1 Method

We focussed our attention to measuring the performance of the RDSS during a node addition operation and during a node removal operation. We began with a stable system of 2 nodes where the two nodes were collectively storing 5000000, 10000000 and 20000000 documents. The stable system was created by initializing an RDSS system with two nodes. We then added the documents to the system by performing a global document add operation via the distribution server.

Beginning with the stable system of two nodes, we proceeded to introduce new nodes, sequentially, up to a maximum of 12 nodes in the system. At each step, we measured the time required to add the new node. Time was measured from the moment there was a distributed consensus to add the node to the moment the operation was committed. Once the system reached the maximum 10 nodes, we reversed the process and proceeded to "kill" a node, one at a time, until we were left with a stable system of only 2 nodes. Again, for each node removal, the time was measured from the moment a distributed consensus had been reached to remove the node until the operation was committed.

6.2.2 Results

We ran our tests 3 times and averaged the results. Table 6.1 shows the time required to add a node and table 6.2 shows the time required to remove a node. In both cases, the more nodes in the system, the less time required to perform the operation. This makes sense since the amount of work (number of documents) remains the same, but there are more nodes to do the work. The times also show that adding a node requires more time, in general, than removing a node. This is probably due to the fact that our implementation scans each remote secondary application instance sequentially, rather than in parallel.

One thing our timing do not show is the amount of time required to detect the failure of an old node or the introduction of a new node. For the scope of this thesis, we were not interested in these times. Detailed measurements of these times can be found in Alan

Nodes	5000000	10000000
2-3	180	428
3-4	106	275
4-5	69	166
5-6	54	127
6-7	53	106
7-8	54	98
8-9	50	85
9-10	51	75

Table 6.1: Node Addition Overhead Time

10-9	21	48
9-8	24	57
8-7	25	61
7-6	29	72
6-5	36	99
5-4	55	146
4-3	86	213
3-2	3-2	381

Table 6.2: Node Removal Overhead Time

Tran's thesis [19].

6.3 Proof of Concept

To test our concept, we chose MG (Managing Gigabytes) as our target IR application [21]. The first step was to write a software layer that could interact with MG and the RDSS. Essentially, all our application layer did was to accumulate and store any data required as part of an operation. Once the operation was to be prepared, our application would build a new input data source and then invoke the MG build tools using the new data source as input.

6.3.1 Method

To test our concept, we used a document collection consisting of 1528595 documents whose size was 4.5 GB. We initialized a distributed group with 2 nodes storing an empty document collection. We then added the document collection using the distribution server. Once the doucment collection was added to the distributed system, we then proceeded to sequentially add a third, fourth and fifth node to the distributed group. Once the distributed group had five members, we sequentially removed nodes until the only remaining nodes were the original two members. This test was run at a different time than our first set of tests. The reason for the smaller test size represents the smaller cluster that was available at the time.

6.3.2 Results

We measured the time required for each operation to occur. These measurements represent the overhead of the RDSS, the application layer that was written to make MG adhere to our application protocol as well as the time required by MG to build the document collection. Table 6.3 shows the results of our tests.

Operation	Time (mm:ss)
Add Document Collection	43:30
Add 3^{rd} node	46:28
Add 4^{th} node	37:07
Add 5^{th} node	30:09
Remove 5^{th} node	38:15
Remove 4^{th} node	44:51
Remove 3^{rd} node	62:13

Table 6.3: MG Application Times

Chapter 7

Future Work

Bibliography

[1] E. Bertino, B.C. OOI, R. Sacks-Davis, K.L. Tan, J.Zobel, B. Shidlovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems*. Addison-Wesley, 1987.

[2] W.J. Bolosky, J.S. Barrera, R.P. Draves, R.P. Fitzgerald, G.A. Gibson, M.G. Jones, S.P. Levi, N.P. Myhrvold, and R.F. Rashid. The tiger video server. *International Workshop on Network and Operating System Support for Digital audio and Video (NOSSDAV)*, 6, April 1996.

[3] 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.

[4] E. W. Brown, J.P. Callan, and W.B. Croft. Fast incremental indexing for full-text information retrieval. volume 20, pages 192–202, September 1994.

[5] L.F. Cabrera and D.D.E. Long. Swift: A storage architecture for large objects. volume 11, pages 123–128, October 1991.

[6] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[7] Gordon V. Cormack, Charlie L.A. Clarke, Christopher R. Palmer, and Samula S.L. To. Passage-based query refinement (multitext experiments form trec-6. *Information Processing and Management*, 36:113–153, 2000.

[8] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Paralle and Distributed Systems*, 10(6), 1999.

[9] D.Cutting and J. Pedersen. Optimizations for dynamic inverted list maintenance. volume 13, 1990.

[10] Christof Fetzer and Flaviu Cristian. On the possibility of consensus in asynchronous systems. Technical report, University of California, San Diego, 1995.

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *ACM*, 32(2):374–382, April 1985.

[12] Richard Golding and Elizabeth Borowsky. Fault-tolerant replication management in large-scale distributed storage systems. volume 18, October 1999.

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

[14] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. volume 7, pages 84–92, October 1996.

[15] 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.

[16] R. Srinivasan. Binding protocols for onc rpc version 2. Technical report, Network Working Group, 1995. RFC 1833.

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

[18] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. pages 289–300, May 1994.

[19] Alan Tran. A network management facility for a fault-tolerant distributed information retrieval system. Master's thesis, University of Waterloo, 1998.

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

[21] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers Inc, second edition, 1999.