

Fastpath Optimizations for Cluster Recovery in Shared-Disk Systems

Randal Burns

Department of Computer Science, Johns Hopkins University
224 New Engineering Bldg., 3400 N. Charles St.
Baltimore, MD 21218
randal@cs.jhu.edu

ABSTRACT

We describe the design and implementation of a *clustering service* for a high-performance, shared-disk file system. The service provides failure detection and recovery, reliable end-to-end messaging, and a centralized and recoverable management interface. We implement novel optimizations in the voting protocol that resolves cluster membership. Optimizations allow clusters to form as quickly as possible without introducing livelock or requiring timeout parameters to be tuned carefully. Our treatment includes performance results that quantify the scalability of the system and measure recovery times.

1. INTRODUCTION

We set out to build a service for failure detection and recovery for high-performance, shared-disk file systems that minimizes the downtime associated with node failure. In the process, we developed optimizations to dynamic linear voting (DLV) protocols that allow clusters to recover from failures as quickly as possible. The optimizations also remove sensitivity to the tuning of the timeout parameters used in voting protocols. In this way, timeouts may be set to large, worst-case values in order to account for network latency, packet loss, and slow computers. Large timeouts have no adverse effect on the performance of the algorithms.

Performance during recovery from outages is as important to overall system performance as are runtime metrics. However, the effort spent on optimizing the runtime performance properties of clustered file systems, such as latency, throughput, and degree of parallelism, has not been met with a similar effort in optimizing performance during failure detection and recovery. Our research addresses this directly, enhancing the performance of recovery without changing the correctness or semantics of the underlying protocols.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Supercomputing 2004 (*SC'04*), Pittsburgh, Pennsylvania, USA
0-7695-2153-3/04 \$20.00 (c)2004 IEEE

Group services and fault-tolerant programming practices provide transparent recovery for applications. Recent research trends have pushed group services toward global-scale distribution. This includes building secure groups services [31, 1, 28], hierarchical group services [2], partitionable (as opposed to primary component) group services [23], and improving state and message recovery semantics [14, 27, 23].

In contrast to the current trends in group services, our research focuses on lightweight, high-performance failure detection and recovery suitable for clustered environments. We exchange a number of features, such as tolerance for heterogeneous networks and byzantine failures, in order to achieve these goals. This decision is sound for file systems, which have little state to recover relative to distributed applications. The file system for which this service was designed, IBM Storage Tank [6, 25], consists of shared-nothing parallel file servers.

The optimizations we implement enhance the performance of the dynamic linear voting protocols on which we build our cluster service. DLV [18, 21] is a multi-phase voting protocol [16]. Associated with each phase is a timeout at which point the leader/coordinator of the protocol declares the round over – no more votes are accepted. Our cluster service, and group services in general, use DLV to resolve cluster membership, determining which servers are operational.

In a technique called *fastpath* optimization, our algorithms predict the membership of the next cluster and use this prediction to complete the formation of the next cluster prior to timeout. When every node in the prototype cluster responds in the voting algorithm, the new cluster forms immediately, without waiting for a timeout. Almost always, configuration changes are predictable and this optimization works.

Fastpath optimizations achieve high performance in a robust fashion. Voting protocols rely on a timeout to terminate a round. Tuning such timeouts can be hazardous. Tune them too high and the protocol takes longer to complete. Tune them too low and the timeout expires before all nodes respond, which results in livelock as clusters repeatedly attempt and fail to form. With fastpath optimizations, clusters form as fast as possible – as soon as all communication completes – without regard for the chosen timeout.

Timeouts can be set for worst case behavior, to account for network latency, lost packets, and slow computers. Long timeouts have no deleterious effect on performance during recovery.

2. RELATED WORK

The objectives and environment of the group service of Jahanian *et al* [17] are most similar our work. They describe a failure detection and communication infrastructure for high-performance computing clusters. This group service was used in the Calypso file system [9] and is the basis for IBM's RS/6000 High Availability (HA) product [15]. The GPFS file system also uses IBM's HA product for failure detection and recovery [33]. The Frangipani file system [35] uses the Paxos algorithm [20] to the same effect.

Our cluster service is built upon dynamic linear voting [18]. DLV has been used extensively in replication [21], group service toolkits [4, 24, 32], and database replication [10]. Recently, DLV has been used as the basis for a distributed shared-memory algorithm [22]. Any improvements to the performance of DLV systems benefit all of these applications.

As group services are increasingly applied in wide-area distributed systems, they have increased their semantics, including a focus on security [31, 28, 1], messaging semantics across failures [14, 27], topologies [2, 13], and expanded classes of failure [29]. The techniques we present in this paper can be applied to the view management and group formation stages of group services. The cluster service we describe operates in a much more constrained environment – high-performance clusters – and does not implement many of the improvements of group services for wide-area distributed systems.

3. FAILURE AND RECOVERY MODEL

Our cluster membership protocols provide failure detection and recovery for a shared-nothing cluster of file system servers which store and manage data on a storage area network (SAN). The service was designed for IBM Storage Tank [6, 25], a cluster file system in which servers provide metadata only (file names, security information, attributes, and location of data). The data are accessed directly by clients from shared disks on the storage network. All metadata in the file system is partitioned into *file sets*. Each server has uniform (latency and throughput) access to all disk drives. Servers store all persistent information about metadata on shared disks as well, which facilitates recovery. If a server fails, another server takes over its file sets. The file metadata are read from shared disks. File metadata brought into a consistent state through log recovery [26] and then brought online by the new server.

The high-performance cluster environment dictates both the failure model and the lightweight nature of the membership service. Our membership service addresses non-byzantine failures with partitions. In partitions, connectivity among nodes is symmetric, transitive, and reflexive, which is reasonable in the LAN environment. Our clustering service provides failure detection and recovery, reliable end-to-end messaging, and a centralized and recoverable management

interface. It does not provide application checkpoints or state recovery, because file servers share little state and that state is recovered by the file system itself, not a middleware service layer. The only shared state is the cluster membership and addressing information to locate the file sets held at different servers. The workload consists of short, atomic operations, rather than long running operations with partial failure semantics. Because no (file system) state is shared between servers, the system does not require causal communication [4]. Nodes communicate only for administrative operations and the movement of load from server to server and all communication is marshaled through an elected *master* server.

The incremental scalability of our membership protocols meets the requirements of high-performance clusters as they evolve toward commodity computers and networks. Protocols allow computers to be added to a cluster in an ad-hoc fashion without a-priori knowledge that the computer is to be added or even exists. No longer are systems made of fixed number of nodes on expensive switches. The combination of a shared-disk file system with DLV-based cluster membership allows computing resources to be flexibly assigned, moved, and exchanged among tasks. DLV allows clusters to be expanded or reduced after applications have been deployed. We envision data centers in which servers are shifted among different clusters in response to daily variation or on-demand changes in workload.

4. PROTOCOLS

Underlying all cluster services is a membership protocol implemented using dynamic linear voting [18]. DLV is a quorum protocol that is often used for replication. We construct a cluster agreement protocol by using DLV for voting only (not replication), *i.e.* the only information distributed throughout the cluster members is the membership itself.

The membership protocol implements two functions: (1) to elect a leader of the cluster, which is a central point of management, and (2) to construct a *primary component* – a single set of servers – which includes the leader that will act as a cluster. Any computer in the majority participates in file serving and non-member computers cease all operation.

DLV helps achieve our scaling goals – incremental growth and shrinkage of a cluster over time. In DLV, a majority is defined as a majority of the previous membership. The protocol maintains no known universe of computers. As computers fail, the membership shrinks. As computers recover, the membership grows. Figure 1 displays the operation of DLV during a series of system events. In this example, a network partition produces two groups of four. *ABCD* becomes the primary component in preference to *EFGH*, because DLV uses lexicographic ordering of nodes to break ties (this is the linear aspect of DLV). The failure of node *A* demonstrates that DLV allows for primary component clusters that are not simple majorities of all available computers. When the partition recovers, nodes *EFH* regain connectivity and contact and join computers in the current primary component cluster.

In addition to the cluster membership protocol, our cluster

System Action	Nodes in Cluster	Non-Cluster Views
All nodes up	{ <u>A</u> BCDEFGH}	
Partition ABCD from EFGH	{ <u>A</u> BCD}	{EFGH}
Node A fails	{ <u>B</u> CD}	{EFGH}
Node G fails	{ <u>B</u> CD}	{EFH}
The partition recovers	{ <u>B</u> CDEFH}	

Figure 1: Example of DLV operating across many system events. DLV always creates a single (primary component) cluster with a single master indicated by the underline.

service provides liveness guarantees and implements heuristics to avoid flurries of network messages seen in cluster recovery [3]. Space constraints prevent us from describing these in detail. Liveness guarantees (and failure detection) are achieved through a heartbeat service, in which cluster servers heartbeat with partners in a ring topology [17]. Flurries of network messages are avoided through a backoff messaging protocol that prevents all computers from attempting to form groups at the same time. Backoff heuristics both (1) allow servers to cede to servers that would make better masters based on lexicographic order and (2) prevent all servers from initiating the master protocol at the same time.

4.1 Master and Subordinate Protocols

Each machine in the cluster runs two threads, a subordinate and master, that execute the cluster DLV protocol. The subordinate thread joins existing clusters, detects failures, and votes for the master. The master thread holds rounds of voting, collects and resolves cluster membership, and distributes membership to subordinates in the cluster. Each node runs a master thread, although only one will succeed in “mastering” the current cluster.

The subordinate thread implements a relatively standard form of DLV, fastpath optimizations have no effect on the state machine or messages. The subordinate state machine includes three states (Figure 2(a)). In the **Active** state, the server participates in a valid cluster – the file system is operational. A server accepts and performs requests on behalf of their clients and communicates with other servers in the cluster. In the **Transition** and **Invalid** states, the file system is inactive, performing no operations on behalf of its clients and sending no messages to other file system servers. In these states, only the cluster membership protocol takes any action, sending messages to other servers in order to locate or construct a valid cluster. The **Transition** and **Invalid** states implement a two-phase voting protocol. In the **Invalid** state, a server is not participating in a cluster, nor has it currently voted (agreed to join) another cluster. In the **Transition** state, a server has already agreed to join a master and will not respond to other requests until the cluster forms (**commit**) or fails (**abort**).

External actions determine the transitions between subordinate states. Figure 2(a) also shows the messages that drive the subordinate from state to state. In the **Invalid** state, the subordinate receives **ping** messages from potential masters. The master uses the **ping** messages to determine the set of alive servers. A subordinate responds to all pings and remain in the **Invalid** state. When the subordinate receives a **membership** message, it tentatively joins the proposed cluster

and enters the transition state. The subordinate joins only a single cluster at a time, awaiting a **commit** or **abort** outcome to the cluster. The membership message includes a membership view, a list of all servers in the proposed cluster, which the subordinate installs. A **commit** message tells the subordinate that a cluster has been formed. The subordinate enters the **Active** state and starts all workload. Alternatively, an **abort** message tells the subordinate that the proposed cluster failed to form and the subordinate returns to the **Invalid** state. The subordinate remains in the **Active** state until a failure occurs or the cluster needs to be modified, because a server is being added or removed. The subordinate receives either an **end** message from the master or a **HB.alert** message from another subordinate when a heartbeat error occurs, indicating a possible server failure.

The master state machine is a standard base implementation of DLV with fastpath optimizations included to speed transitions among states. We first describe the base implementation of DLV and then discuss the specific operation of fastpath optimizations in Section 4.2. The master state machine has four states (Figure 2(b)). In the **NotMaster** state, the master does nothing. In the **Forming** state, the master contacts subordinates using the **ping** message, hoping to find a quorum of servers with which it can form a cluster. In the **View** state, the master has constructed a potential cluster, sent the **membership** message to these servers, and awaits the votes of the members. In the **Master** state, the master sends the **commit** message to all subordinates. After that, it awaits administrative requests or network events that will end the cluster.

The cluster membership protocol drives the master state machine and generates the messages that drive the subordinate state machine. We now describe the base protocol, leaving fastpath optimizations for later discussion. In the **NotMaster** state, the master thread waits for timer and when the timer elapses it attempts to form a cluster if it is an *eligible* master. To be eligible, the server must (1) not be a (subordinate) member of a group currently, (2) have participated in the last known group, and (3) be a feasible master lexicographically. In DLV, only servers in the lower half (lexicographically) of the last membership can possibly master the next group. In the forming state, the master awaits **ping** responses and includes any servers that respond as potential members. A timeout ends the **Forming** state and the master progresses to the view state. In the view state, the master awaits **membership** responses. If the master receives a DLV majority of responses, it commits the membership locally and enters the **Master** state. If a majority is not reached, the master sends **abort** messages to the potential members

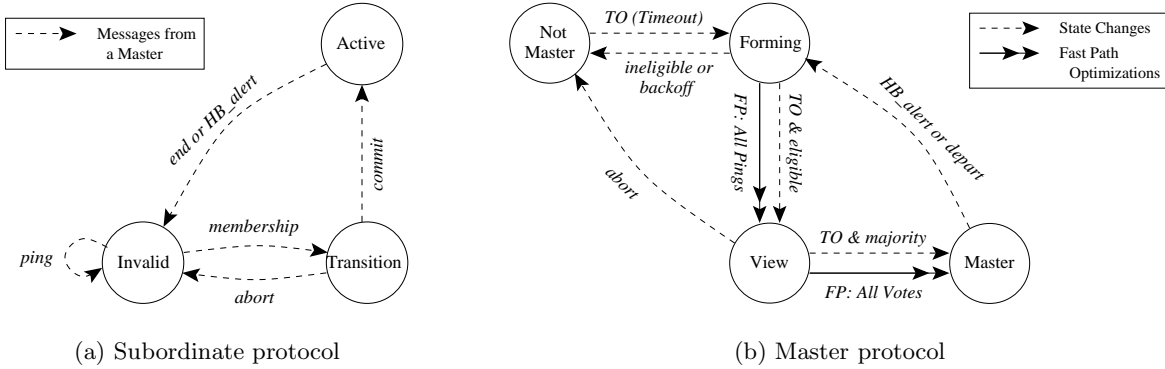


Figure 2: Protocols state machines for cluster membership of the Master and Subordinate threads at each server.

and returns to the **NotMaster** state. When the **Master** state ends, the master returns immediately to the **Forming** state, because it expects to be the master of the next group as well. This optimism is related to the lexicographic naming of servers, in which the master of the previous membership becomes the master of the subsequent membership.

The protocol has the same blocking problems as does two-phase commit [34, 19]. The protocol could be made non-blocking [24, 20] at the expense of another round of communication, *i.e.* a third phase. We deemed this an undesirable trade-off. The conditions under which our protocol blocks are exceedingly rare – the same conditions under which two-phase commit protocols block, in which both the master and at least one more node fail without any of the alive nodes discovering the protocol outcome. Our system recovers from blocking by announcing the problem through administrator alerts. This works well in administered LAN environments. This is the same way in which we recover from the unlikely event that the cluster service failed to form a majority – a possibility in all DLV systems [16].

4.2 Fast-Path Optimizations

The protocol implements several optimizations that improve the performance of cluster membership protocols. These optimizations allow the DLV membership protocol to complete as fast as possible, prior to the timeout for each round. Fast-path optimizations take advantage of the fact that most cluster membership changes can be predicted accurately. When the master correctly predicts the membership of the next cluster and all servers in that prediction have responded, the master advances the protocol state without waiting for a timeout. Fast-path cluster formation makes minor changes to DLV that improve operation without changing the correctness or semantics of cluster membership protocols.

In the fastpath transition from the **Forming** to the **View** state (Figure 2(a)), a prospective cluster master constructs and matches a predicted *prototype* cluster. When the previous cluster terminates, the master constructs and stores a prototype cluster which consists of the servers expected to be members of the next cluster. The master sends out **ping** messages and awaits replies from active subordinates

that will participate in the next cluster. As **pingResponses** come in, the Master compares the set of servers that have responded to the prototype cluster. When the responding servers match or exceed the prototype cluster, the master advances the protocol state to **View**. This is called the **AllPings** condition, because all expected **pingResponses** have been received.

The fastpath transition from **View** to **Master** operates similarly, but has fewer cases and simpler semantics. In the **View** state, the master has constructed a set of servers that it thinks are active. It sends them **membership** messages asking the subordinates to vote on the cluster membership. The prototype cluster for this transition is exactly the same set of servers that responded to **ping**. The fastpath transition works as long as all servers that were active in the previous round are active in the voting round.

Masters accurately predict membership in the next cluster for all administrative membership changes and most failure cases. When servers are added or removed, the prototype cluster is simply the current membership plus or minus the added or removed servers respectively. During simple failures of a single server, the current cluster ends due to a communication or heartbeat failure that is reported to the master. The master excludes from the prototype cluster the server that experienced the failure. Table 1 analyzes the applicability of optimizations under different system events.

DLV’s flexible membership makes the process of predicting subsequent cluster membership non-trivial, particularly in complex configuration changes affecting many computers at once. We use a simple heuristic that allows fastpath optimizations to work when many computers recover simultaneously. The master sends **ping** messages to all known computers, not just computers that participated in the previous cluster. This includes servers that have failed and not responded for a while. This policy attempts to include as many active servers as possible with the goal of constructing as large a cluster as possible. It also reduces the number of times cluster membership protocols get invoked. For an example, we consider a network partition that has caused several computers to be excluded from the primary component. When the network partition repairs, one of the omit-

System Event	AllPings	AllVotes	Behavior
Single server fails	Yes	Yes	Failure is detected by a peer node. Failed node is omitted from the next prototype cluster.
Server(s) decommissioned	Yes	Yes	When one or more servers are removed, the prototype cluster omits those servers and forms a cluster with the remaining servers.
Single server recovers	Yes	Yes	Recovering server contacts the Master and is added to the prototype cluster.
Server(s) commissioned	Yes	Yes	Multiple servers are added to the prototype cluster and a new cluster forms.
Multiple servers fail	No	Yes	The failure of one server causes the cluster to reform. The prototype cluster is not correct for AllPings , but AllVotes is evaluated against only those servers that respond to AllPings .
Multiple servers recover	Often	Yes	Pings are sent to all failed servers. When failed servers respond in a timely fashion, AllPings works. See text for details.
Failure in round 1	No	Yes	A member of the prototype cluster fails, preventing AllPings . AllVotes proceeds based on the responding servers.
Failure in round 2	N/A	No	AllPings is not relevant. AllVotes fails because a member of the prototype cluster has failed.

Table 1: Summary of the performance of DLV optimizations organized by failure/recovery scenario.

ted servers attempts to join the cluster, causing the master to run the membership protocol. By pinging all of the failed servers, the protocol reintegrates the whole partition in a single round. However, sending out **pings** to computers in an unknown state may make it harder to construct accurate prototypes.

Fastpath formation operates in more exceptional failure circumstances, including transient network errors and the cluster master failing. A server who was not master of the previous cluster constructs a prototype cluster also. However, in the prototype, it leaves out all servers lexicographically lower than itself in the previous cluster, because any lower server would be master in preference to itself. When the master fails, the next server becomes master and fastpath cluster formation works. Fastpath also works during many transient network errors, in which servers appear to fail and immediately recover. A server that appears to fail will be left out of the prototype cluster. However, this server receives a **ping**. The server that experiences a transient error gets into the next cluster as long as it send its **pingResponse** prior to **AllPings**. This race condition between the **pingResponse** and **AllPings** has no correctness implications. If the server does not make it into the membership, it joins the next cluster. The only cost is that the protocol must be run again.

Fastpath optimization uses several message ordering heuristics to minimize race conditions between messages and **AllPings**. When a server fails, we send a **ping** to it first, so that if it is alive, it will “be ahead” of other servers in sending the **pingResponse**. Similarly, when a failed server is recovering, we send **ping** messages to all known failed servers prior to

pinging all servers in the current membership. This helps when a network partition merges that was isolating many computers from the membership. The protocol only sees a single server recover; it starts the membership protocol in response. Other servers may have recovered also and the protocol **pings** them first to get them ahead of the **AllPings** deadline.

Fastpath fails to work when several servers fail at once. When heartbeats or messaging errors detect the first failed server, the master constructs a prototype that does include that server, but does include any other failed servers. When this happens, **AllPings** never occurs and the protocol waits for a timeout to end the round and advance state. These types of failures arise mostly when network partitions occur, isolating several servers at once.

5. DISCUSSION

Many practicalities govern the implementation of failure detection and recovery services. Efficient and sound protocols are only a building block. Issues that we address in our system include (1) the construction of a centralized management interface on a decentralized cluster with flexible membership and (2) achieving safety guarantees on the shared-disk networks in addition to the server-to-server communication network.

5.1 Cluster Management

Providing centralized management in computing environments made of changing resources is a challenge in distributed and cluster computing. Centralization keeps the

management abstraction simple and provides a single operational view of a system. This ultimately lowers the cost of owning and operating a system. However, cluster and distributed systems consist of a changing set of peers and, by definition, do not have centralized services.

The presented cluster membership uses DLV to provide highly available centralized management among peers of computers in a cluster. We place a single management interface on the elected leader of a primary component. The interface allows administrators to monitor and update global information about the cluster. This includes: servers in the cluster, servers not in the cluster that were in recent clusters, the workload map (assignment of file sets to servers), and server performance statistics. It also allows the system to compile and report aggregate information about all servers. The management interface is available through any server in the cluster; servers take administrative requests and forward them to the cluster master. In this way, as the cluster evolves and the master changes, administrative clients do not need to track or relocate the master.

Servers outside of the current cluster membership view need to alert administrators, but do not have access to the management interface through the master. While the cluster master may report on servers that have failed, this is not sufficient. Non-cluster servers must notify administrators when DLV fails to build a primary component. In this case, there is no management interface, because there is no master. Furthermore, this is the most important failure to report. Our approach is to have a server emit a periodic SNMP alert whenever it believes it should be a member in a cluster, but it cannot find a primary component to join. Failure to form a cluster results in a flurry of notices requesting repair. A network partition has the same results, which is desirable if the administrator is on the same side of the partition of the failed servers.

5.2 On the Dangers of Shared-Disk Systems

The described cluster membership protocol does not by itself provide all of the needed safety guarantees. While it protects all communication among cluster nodes, it cannot ensure actions between servers and shared disks. The problem lies not in the design of the cluster service. Rather, it is a problem fundamental to shared-disk systems.

Cluster membership does not always protect the integrity of the data on shared-disks. In the cluster, disks are “dumb” entities, in that they are not managed by membership. Disks accept reads and writes without consideration for whether the reader or writer is a member of the cluster. We are concerned with computers outside of the current membership writing data to a disk.

Servers mostly avoid these undisciplined writes to shared disks by validating their membership prior to issuing any I/O request. This approach would be sufficient under a couple of (unreasonable) assumptions: (1) I/O requests complete in a timely fashion and (2) servers exhibit failstop behavior. I/O requests need to complete so that when a computer falls out of a cluster its outstanding I/O finishes prior to the next cluster starting to do I/O to the same

data. Computers must be failstop so that a computer that exhibits a network outage does not subsequently submit an I/O request.

Neither of these assumptions are reasonable. Computers are complex systems that fail in complex ways. In particular, the I/O subsystems of a server is outside the control of the cluster membership protocol. The device drivers, HBAs, and network hardware that implement I/O may delay or queue I/O requests indefinitely.

Other methods for protecting the integrity of data are needed. A technique called *fencing* [7] allows the system to configure connectivity between hosts and storage. When a computer fails and falls out of the cluster, the master of the primary component “fences” the failed computer from all storage. Currently, fencing can be implemented by zoning techniques in SAN fabrics, such as Brocade SecureOS [5] or through access control and capability systems that are used primarily for security [12, 30, 8].

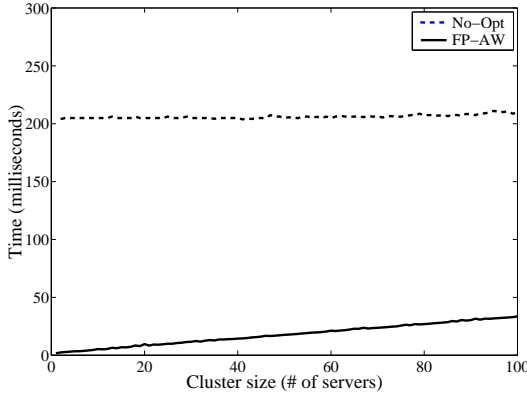
Some research has attempted to address uncontrolled writes by including the shared disks as a participant in membership protocols. Examples include disk Paxos [11] and asymmetric multicast protocols [29]. In these systems, disks hold a membership view of valid servers and discard I/O requests from non-members. However, these approaches are not widely applicable, because few storage devices are programmable and provide the necessary computing resources to run such protocols. We prefer the use of fencing support provided by storage networks and devices.

6. EXPERIMENTAL RESULTS

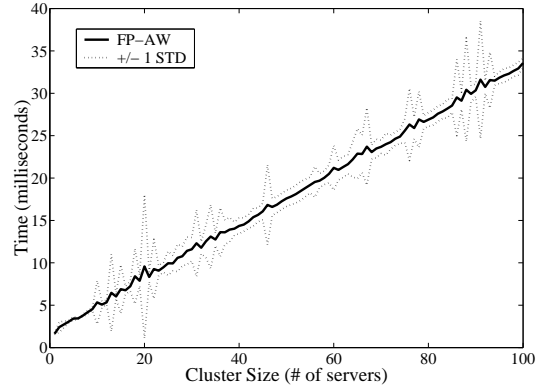
We conducted experiments on our implementation of the cluster service in order to evaluate absolute performance and determine the benefits of fastpath optimizations. Experiments were conducted using 5 computers in an IBM x330 series cluster, running RedHat Linux 7.3 with the 2.4.19 SMP kernel. Each machine is identically configured with dual 1.3 GHz Pentium III processors, 1.25 GB of RAM, and a IBM Ultra2 18.2G, 10K rpm SCSI drive. A 100Mb/s Ethernet switch connects the cluster nodes. All computers have symmetric access to shared disks, an IBM FastT 200 RAID array of 74.3 GB IBM Ultra2 10K rpm disks, connected by a Fibre channel network using a IBM 2109 model F16 switch at 1 Gb/s.

Fastpath optimizations have several benefits which include reducing the time required to form a new cluster, making cluster formation performance insensitive to the selection of a timeout parameter, and reducing the performance variance of the algorithm. Figure 3(a) compares the performance of cluster formation with and without fastpath optimizations. This experiment is based on the *All-Write* policy – the most pessimistic measure of our algorithms’ performance. The cluster formation algorithm uses a timeout value of 100 ms on each round of communication. Optimizations reduce the time to form the cluster from roughly 200 ms – the time to wait for two timeout cycles during the protocol – to less than 35 ms for 100 cluster nodes.

Fastpath optimizations allow the time to form clusters to be



(a) Comparing the time to construct a cluster with and without fastpath optimization.



(b) A standard deviation.

Figure 3: The performance of the *All-Writers* variation. Each data points represents an average of 40 trials.

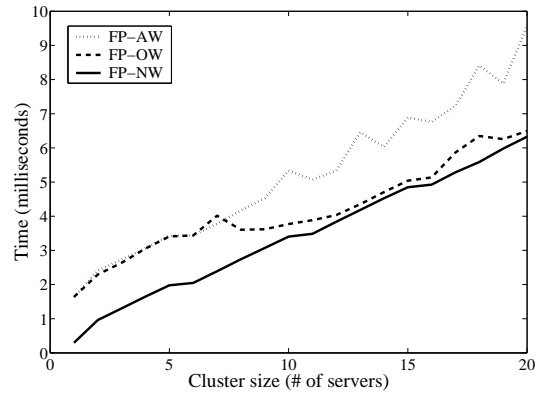
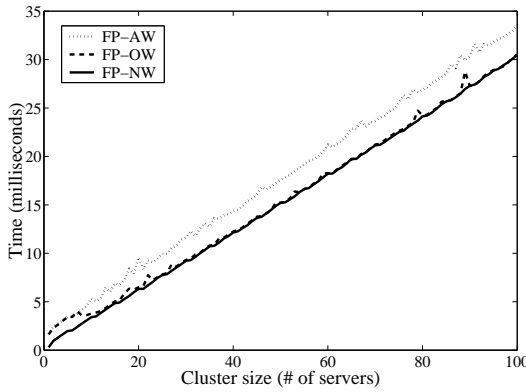


Figure 4: Time required to complete fastpath cluster formation. These data compare the variants of mapping virtual nodes to physical computers: *All-Writers* (FP-AW), *One-Writer* (FP-OW), *No-Writer* (FP-NW). The graphs depict the same data at different scales. Each data point represents an average of 40 trials.

minimized without exposure to livelock. One approach to increasing performance is to tune down the timeouts to complete cluster formation more quickly. The timeouts could even be tuned adaptively based on the number of cluster nodes in the proposed groups. This can move the *No-Opt* line down toward the *FP-AW* line. However, the fastpath optimization represents a lower-bound on how quickly communication can finish. If we tune the timeout down below optimization line, then cluster formation does not finish prior to the timeout expiring and the algorithm backs off and tries to form the cluster again.

Performance variance makes it hazardous to tune down an absolute timeout value. The chosen timeout must be set significantly higher than the fastpath value to account for variance and avoid repeating the membership protocol, which can lead to livelock. Figure 3(b) depicts the width of the distribution of completion times using an experimental standard deviation around the sample mean of *FP-AW*. Fastpath formation exhibits a high-degree of variability in completion time. Any timeout would need to be tuned several standard

deviations above the mean to avoid frequent retries.

A key issue in our experiments is how to faithfully evaluate cluster formation and fastpath optimizations on large clusters. Our approach is to run multiple instances of the cluster service on each computer. The problem with this approach is that the multiple *virtual* nodes on a physical machine interact with each other. This interaction hinders performance in some cases, *e.g.* thread scheduling and I/O contention. In other cases, it increases response time, such as sending network messages between nodes on the same computer. We organize virtual nodes so that whenever possible they communicate with virtual nodes on another computer. We also restrict the number of nodes running on a computer to twenty. Beyond that number, we find that the system does not schedule all threads in a timely fashion.

Most interference between virtual nodes on the same host arises during I/O. In DLV it is necessary for each machine to harden (make persistent through I/O) the votes it has cast and the current protocol state. Fastpath optimizations

do not introduce new I/O requirements. Each subordinate performs I/O synchronously prior to responding to a **Group** message and asynchronously prior to responding to a **Commit** or **Abort** message. Furthermore, each subordinate submits this I/O to disk at approximately the same time, resulting in artificial queuing delays when simulating multiple nodes on a single physical computer.

We use three slightly different approaches when mapping virtual nodes onto physical machines that attempt to provide upper and lower bounds on the performance of cluster formation. In the *All-Write* variation, all nodes perform all I/O. The *One-Writer* variation has a single node on each physical host performing I/O. Other nodes run the clustering protocol without hardening any state. In the *No-Write* variation, no I/O are performed.

Examining the different policies are of interest because they shed light on the factors that limit performance and scalability. The variants of the algorithm are not interesting in and of themselves. In particular, we reveal that for small clusters – fewer than 5 nodes – I/O limits completion time. As we move to large clusters, the time to complete the network messages of the protocol dominate all other factors.

Figure 4 shows the relative performance of the three policies for mapping virtual nodes onto physical machines. In particular, the difference between the three data sets describes the effect of I/O on performance. While I/O does degrade performance increasingly for larger clusters, it is a secondary performance factor. Network communication during protocol rounds dominates. The *No-Write* policy provides a lower-bound approximation on the execution time of clusters of 100 nodes. There is no interference between virtual nodes because there is no I/O. This virtual node configuration has slight advantage over a real configuration, as it sometimes sends network messages to virtual nodes on the same physical machine. The *All-Write* policy provides a conservative estimate of cluster performance. For a small number of nodes, performance is limited by I/O. However, for more than 10 nodes, network performance dominates. Therefore, optimizing the messaging protocol directly improves the performance of the cluster service. In the *One-Writer* policy, the performance limiting factor shifts from I/O to network. For 5 or fewer servers, the *One-Writer* policy is identical to the *All-Write* policy and tracks the performance very closely. As it moves to larger numbers of servers, network factors increase in importance and the *One-Writer* policy more closely follows the *No-Write* policy. I/O overhead can be measured as the difference between the curves for the *All-Write* and *No-Write* policies. I/O overhead is low, less than 2 ms for a single virtual node per computer, even though writes are performed synchronously to disk. The service writes small data – about one disk block – and always writes to the same region of disk. The disk head is well positioned, which prevents seeks.

Fastpath optimizations make the cluster membership portion of failure recovery fast. Figure 4 shows that 100 nodes clusters form a new membership in less than 35 milliseconds. For comparison, recovery also involves reading file sets from shared disks (Section 3), which we estimate to take between 200 milliseconds and 5 seconds, depending on the data set

and how it failed.

7. CONCLUSIONS

We have described fastpath optimizations to dynamic linear voting protocols that are used to speed cluster recovery in a shared-disk file system. Fastpath optimizations remove the need to attempt to accurately tune timeout parameters. Clusters form as fast as possible at any timeout value. The timeout value can be set for worst case latency, without degrading performance in practice. Fastpath optimizations do not change the correctness or semantics of DLV recovery protocols. These techniques apply to the many other applications of DLV, such as database replication, group service toolkits, and distributed shared memory.

8. REFERENCES

- [1] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 2000.
- [2] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS-98-4, Center for Network and Distributed Systems, Johns Hopkins University, 1998.
- [3] M. L. G. Baker. *Fast Crash Recovery in Distributed File Systems*. Ph.D. dissertation, University of California at Berkeley, 1994.
- [4] K. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.
- [5] Zoning implementation strategies for brocade SAN fabrics. Brocade Inc., White Paper, 2002.
- [6] R. Burns. *Data management in a distributed file system for Storage Area Networks*. Ph.D. dissertation, University of California at Santa Cruz, 2000.
- [7] R. Burns, R. M. Rees, and D. D. E. Long. Safe caching in a distributed file system. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2000.
- [8] D. Naor *et al.* Object store security document. Storage Networking Industry Association (SNIA), 2003.
- [9] M. Devarakonda, D. Kish, and A. Mohindra. Recovery in the Calypso file system. *ACM TOCS*, 14(3), 1996.
- [10] A. el Abadi and S. Toueg. Maintaining consistency in partitioned, replicated databases. *ACM Transactions on Database Systems*, 14(2), 1989.
- [11] E. Gafni and L. Lamport. Disk paxos. In *Proceedings of the International Symposium on Distributed Computing*, 2000.
- [12] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. Ph.D. dissertation, Carnegie Mellon University, 1999.

- [13] L. Gu and J. G.-L. Aceves. New error recovery structures for reliable networking. In *Proceedings of the International Conference on Computer Communications and Networking*, 1999.
- [14] K. Guo, W. Vogels, and R. van Renesse. Structured virtual synchrony: Exploring the bounds of virtually synchronous group communication. In *Proceedings of the ACM SIGOPS European Workshop*, 1996.
- [15] RS/6000 SP high availability infrastructure. IBM Redbook SG224-4838, IBM, 1996.
- [16] K. Ingols and I. Keidar. Availability study of dynamic voting algorithms. In *Proceedings of the International Conference on Distributed Computing Systems*, 2001.
- [17] F. Jahanian, R. Rajkumar, and S. Fakhouri. Processor group membership protocols: Specification, design, and implementation. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 1993.
- [18] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15, 1990.
- [19] I. Keidar and D. Dolev. Increasing the resilience of atomic commit at no additional cost. In *Proceedings of the Symposium on the Principle of Database Systems*, 1995.
- [20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), 1989.
- [21] D. D. E. Long and J. F. Pâris. Efficient dynamic voting algorithms. In *Proceedings of the International Conference on Data Engineering*, 1988.
- [22] N. Lynch and A. A. Schwartzmann. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the International Conference on Distributed Computing Systems*, 2002.
- [23] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), 1996.
- [24] C. Malloth and K. Schiper. View synchronous communication in large scale networks. In *Workshop of the ESPRIT project BROADCAST*, number 6360, 1995.
- [25] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM storage tank: A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2), 2003.
- [26] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), 1992.
- [27] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *The IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1994.
- [28] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable Corba applications with the Immune system. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 1999.
- [29] J. Palmer, R. Strong, and E. Upfal. Nonblocking ordered reliable multicast in an unreliable distributed environment. Technical Report RJ-10096 (91913), IBM Research Division, 1997.
- [30] B. C. Reed, D. D. E. Long, E. G. Chron, and R. C. Burns. Authenticating network attached storage. *IEEE MICRO*, 20(1), 2000.
- [31] O. Rodeh, K. Birman, and D. Dolev. The architecture and performance of security protocols in the Ensemble group communication system. *ACM Transactions on Information and System Security*, 4(3), 2001.
- [32] L. Rodrigues and P. Verissimo. xAMP: A protocol suite for group communication. Technical Report RT/43-92, INSEC, 1992.
- [33] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technology*, 2002.
- [34] D. Skeen. A quorum-based commit protocol. In *Workshop of Distributed Data Management and Computer Networks*, 1982.
- [35] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the ACM Symposium on Operating System Principles*, 1997.