PhatRaid, yo

Andrew Johnson Daniel King Lucas Waye
Scott Moore

May 13, 2014

Distributed file stores, like Google's Chubby [1] and CS260r's Phat [3], send all files through a single master node which consistently replicates the file on many slaves. In this scheme, the master node's throughput is a bottleneck for storing large files. We present PhatRaid which mitigates this bottleneck by partitioning files across many Paxos groups which form a RAID [5] group.

1 Introduction

Google's Chubby [1] is a distributed file system which achieves consensus in the face of limited failures. Unfortunately, for storing large files disk latency on the cluster nodes is a bottleneck. We propose building a RAID (Redundant Array of Inexpensive Disks) [5] group out of distributed file system clusters. Large files are sharded and disk latency is reduced proportional to number of clusters. Failure resiliency is not completely sacrificed and we provide an analysis of the new failure modes.

Our setting is in a data center where we have low network latency, high network throughput, and the need to dependably store large files with tolerance to some CPU, network, and disk failure. In this setting, we hypothesize that the major bottleneck for replicated file storage is disk read/write latency. We combat this bottleneck with ideas from RAID, which allow for faster file access, and ideas from distributed consensus algorithms, which allow for consistent replication.

1.1 Disk Performance

An important consideration in our project is the impact disk performance has on overall commit latency. We believe that disk read/write latency is the major contributor to storing large files in a distributed system.

In a review of hard disk performance [2], Hard Disk Drives (HDD) were compared with Solid State Drives (SSD). HDDs use a physical magnetic platter that spins so that a drive head can read or write data by sensing or manipulating the magnetic field in a particular position on the platter. In contrast,

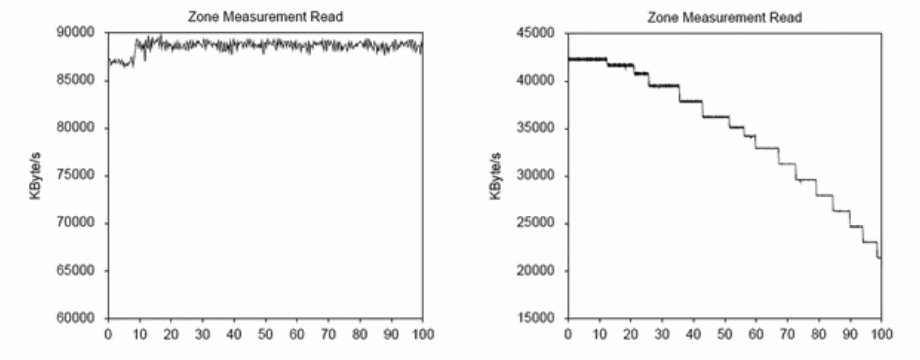


Figure 1: **Left Chart**: Solid State Disk reads across entire drive (Left – Outermost sector, Right – Innermost sector), **Right Chart**: Hard Disk reads across the entire drive (Left – Outer rim, Right – Inner rim). From [2]

SSDs have no moving parts and instead uses integrated circuits as memory to store data persistently. Because the way in which data is accessed is different, the performance characteristics between the two are different.

Figure 1 shows a comparison between read times for a SSD and HDD as a function of position on the disk. HDD storage performance is greatly influenced by where on the disk platter the data was placed. Data can be placed on the outer rim of the platter much more quickly than data placed near the center of the platter as the outer rim is moving faster in comparison (due to conservation of angular momentum). In contrast to HDDs, SSDs do not exhibit this problem as there is no spinning platter. The left diagram in Figure 1 shows relatively constant read performance throughout the entire disk. Additionally, HDDs have difficultly with random accesses as the drive may need to wait for the drive head to be directly over the correct position on the platter. This problem becomes especially apparent when an individual file is fragmented into multiple blocks by the file system. SSDs have better random access times, in general, than HDDs. Sequential access tends to be many times faster than random access for both SSDs and HDDs [2]. HDD performance can also can suffer from *spin-up delay*, which happens as a result of a power-saving measure that stops spinning the disk during idle times.

We decided that HDDs have far too much performance variability in order to be properly modeled. As a result, we instead chose to simulate the performance characteristics of SSDs as they have relatively constant access times regardless of sector position on the drive and also do not suffer from *spin-up delay*. Also, because HDDs are slower, and our hypothesis supposes that disk is the bottleneck, this should be a more challenging case for PhatRaid.

Figure 2 shows the performance for SSDs from a 2013 survey of commer-

Solid State Disk Performance Benchmarks	
Sequential Write Speed	55.03 - 481.85 MB/sec
Write Access Time	0.03 - 0.32 ms
Sequential Read Speed	207.95 - 522.45 MB/sec
Read Access Time	0.03 - 0.22 ms

Figure 2: Solid State Disk performance for middle 95% of solid state drives tested in a benchmark. Benchmark available at Benchmark available at `http://www.tomshardware.com/charts/ssd-charts-2013/benchmarks%2C129.html`.

cially available SSDs. Sequential write speeds measure the throughput (in megabytes per second) of data stored on the drive when the data is being written in the drive's natural sequential order (e.g. increasing sectors). Similarly, sequential read speed measures the throughput for sequential reads. Read/Write access times represent the latency of performing a single random read or write.

2 Implementation

2.1 Erlang Primer

PhatRaid is implemented in Erlang. Erlang is a pure, functional language with extensive library support for distributed applications. Erlang has one unit of concurrency, the *process*, and one unit of distribution, the *node*. Each Erlang node consists of an arbitrary number of processes. Erlang nodes may live on the same machine or on different machines connected by a network. Every Erlang node is given a name by an Erlang name server. Erlang processes may communicate with any Erlang node or process it can name. Erlang processes interact with one another by making RPC calls or sending messages. Identifiers beginning with a lowercase letter are actually *atoms*, a literal constant similar to Scheme and Lisp's symbols. Identifiers beginning with an uppercase letter are normal identifiers.

2.2 System Organization

PhatRaid is implemented in Erlang, reusing our previous implementation of Phat. The implementation is divided into three domains: the server cluster, the client, and the raid-client. The server code implements a variant of the Viewstamped Replication (VR) [4] consensus algorithm as well a file system.

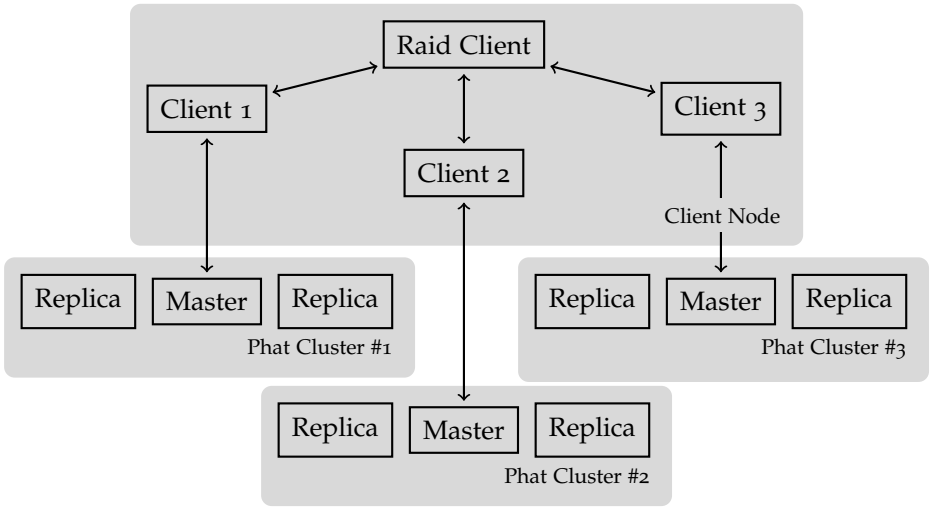


Figure 3: A PhatRaid(3,1) Group

The client code exposes a file system API which hides the communication bookkeeping. The raid-client provides a file system API which transparently partitions files and reassembles them as they are sent and received from the server cluster.

Figure 3 depicts the organization of our system. The raid-client delegates communication to each client. Each client is connected to one Phat cluster, which has a master and a number of replicas.

The master and each replica in a Phat cluster maintains a separate file system. Two different phat clusters do not communicate directly. The raid-client and client processes all live on one Erlang node. Each Phat master and Phat replica lives on its own Erlang node along with its file system.

2.3 Disk Emulation and Abstraction Layer

We developed a novel *"Disk Emulation and Abstraction Layer"* (DEAL) to simulate SSD performance for our evaluation. The DEAL simulates read and write delay using the information from Figure 2. We calculate disk delay as:

$$\text{ReadDelay}(x) = x/\text{SeqReadSpeed} + \text{ReadAccessTime} \quad (1)$$

$$\text{WriteDelay}(x) = x/\text{SeqWriteSpeed} + \text{WriteAccessTime} \quad (2)$$

where x is the size of the file. During filesystem operations, these formulas are used to put the process to sleep in proportion to the size of the file being written to or read from the drive.

Instead of sending very large files, we sent smaller strings that were treated by the DEAL as being much larger (e.g. 1 character represented 1MB of data). As a result, we extended the DEAL to perform an additional delay for performing bitwise XOR on a file as that was the only other operation on a file for our tests. In our model, drive delays were much greater than XOR delays (e.g. 20 GB/second XOR throughput versus 200 MB/second drive throughput).

2.4 RAIDing Phat

A PhatRaid(C,f) implementation is parameterized by two numbers: C, the number of Phat clusters, and f, the Phat cluster failure tolerance. A single Phat cluster contains $2f + 1$ nodes. A PhatRaid group contains C clusters for a total of $C(2f + 1)$ Phat nodes. A PhatRaid group can be concurrently accessed by an arbitrary number of raid-clients.

On a raid-client node, C client processes are spawned each of which maintains a connection to one of the C Phat clusters. Each client process mediates communication between its assigned Phat cluster and the raid-client.

A store operation on the raid-client breaks the file into $C - 1$ chunks, stores each chunk on a cluster, and stores a parity bit on the remaining cluster. A raid-client fetch operation reconstructs the file from the $C - 1$ chunks, using the parity bit if necessary.

2.5 Phat Clusters

A server cluster is a collection of nodes which maintain a distributed file system. Each server cluster designates a unique master node (aka the Phat Master). The other nodes are known as Phat replicas or just replicas.

The master and each replica have four components: the supervisor, the server, VR, and the file system:

- the supervisor – restarts the other three processes if any dies
- the server – passes messages to the VR process if this node is the Phat master; otherwise, it drops the message and informs the client of the current master node
- VR – maintains a log of opaque messages, achieving consensus with other nodes in the cluster via the Viewstamped Replication protocol

- the file system – maintains a hierarchical file system with locks

2.6 Erlang Behaviors

The four components are implemented using Erlang behaviors. An Erlang behavior is a framework which implements common patterns like servers and finite-state machines. In particular, the supervisor uses the `supervisor` behavior, the server and file system both use the `gen_server` behavior, and VR uses the `gen_fsm` behavior.

- The `supervisor` behavior provides a framework for restarting failed processes.
- The `gen_server` behavior provides a framework for many-client single-server interactions. We implemented functions to process messages formatted as Erlang tuples. The `gen_server` behavior handles low-level socket interaction, messaging queueing, etc.
- The `gen_fsm` (finite-state machine) behavior generalizes the `gen_server` behavior by permitting the server to have a finite number of states. Each state has a set of functions with which to process messages. This behavior can be seen as an ad hoc polymorphic variant of the `gen_server` behavior.

Additionally, the `gen_server` and `gen_fsm` behaviors allow the programmer to specify an arbitrary state value which will be passed around à la functional reactive programming. We call this state value the *store*.

2.6.1 The `supervisor` Behavior

Listing 1 is taken from our supervisor code. When a supervisor is started, the `supervisor` behavior looks for a procedure called `init`. The `init` procedure returns a child specification, which is a pair of a restart specification and a list of children. In Listing 1, the restart specification, `{one_for_all,1,5}` specifies that if any one node dies, all nodes should be restarted, unless more than one failure has occurred in the past five seconds. If more than one failure occurs in five seconds, the supervisor kills all its children and then shuts itself down.

Each child in the list of children is a 6-tuple consisting of: child name, initial procedure call, child transience, shutdown style, child type, and necessary modules.

- The initial procedure call is a 3-tuple of module, procedure name and argument list.

```
6  init([Master|Rest]) ->
7      Node = node(),
8      VRS = lists:map(fun (N) -> {vr,N} end, [Master|Rest]),
9      {ok, {{one_for_all, 1, 5},
10           [ { fs
11             , {fs, start_link, []}
12             , permanent, 1, worker, [fs]}
13           , { ps
14             , {server, start_link, []}
15             , permanent, 1, worker, [server]}
16           , { vr
17             , {vr, startNode, [{vr,Node}, VRS, fun server:commit/3]}
18             , permanent, 1, worker, [vr]}]}}.
```

Listing 1: The `supervisor` Behavior — `phat.erl`

- Every child's transience is specified as permanent, meaning they *should* be restarted.
- Every child has shutdown style 1 meaning they will first be asked to terminate and one second later they will be forcibly terminated[1].
- All of our children are workers, which are distinguished from sub-supervisors.
- The necessary modules are dynamically loaded when the child processes is initialized. In PhatRaid, each child only depends on one module.

2.6.2 The `gen_fsm` Behavior

A state in the `gen_fsm` behavior manifests as a procedure which accepts two arguments: the incoming message and the store. The return value of a state-procedure is usually the 4-tuple {next_state,NextState,NewStore,Timeout}. The `gen_fsm` behavior will transition to NextState and set the store to NewStore. When a new message is received the `gen_fsm` behavior will invoke the NextState procedure with the new message and the NewStore.

The Phat file system reaches consensus via the Viewstamped Replication protocol [4]. Listing 2 defines how to handle prepare messages sent to a node in the replica state. It is piecewise defined by pattern matching on the arguments. The first two definition clauses use the when keyword to specify arbitrary required relationships between the matched variables.[2]

[1] We could also have specified the hastier shutdown style: brutal_kill

[2] ?debugFmt is a macro for printing debug messages when a flag is set

```
176 replica( {prepare, MasterViewNumber, _, _, _, _, _}
177        , State = #{ viewNumber := ViewNumber})
178     when MasterViewNumber > ViewNumber ->
179     ?debugFmt("my view is out of date, need to recover~n", []),
180     startRecovery(State);
181
182 replica( {prepare, MasterViewNumber, _, _, _, _, _}
183        , State = #{ timeout := Timeout, viewNumber := ViewNumber})
184     when MasterViewNumber < ViewNumber ->
185     ?debugFmt("ignoring prepare from old view~n", []),
186     {next_state, replica, State, Timeout};
187
188 replica( { prepare, _, Op, OpNumber
189          , MasterCommitNumber, Client, RequestNum}
190        , State = #{ prepareBuffer := PrepareBuffer }) ->
191     Message = {OpNumber, Op, Client, RequestNum},
192     NewPrepareBuffer = lists:sort([Message|PrepareBuffer]),
193     processPrepareOrCommit( OpNumber
194                           , MasterCommitNumber
195                           , NewPrepareBuffer
196                           , State
197                           );
```

Listing 2: The `gen_fsm` Behavior — `vr.erl`

```
375 processPrepareOrCommit( OpNumber, MasterCommitNumber, NewPrepareBuffer
376                       , State = #{ timeout := Timeout
377                                  , commitNumber := CommitNumber
378                                  , masterNode := MasterNode
379                                  , myNode := MyNode
380                                  , viewNumber := ViewNumber
381                                  , allNodes := Nodes})
382   when CommitNumber > MasterCommitNumber ->
383     NewViewNumber = ViewNumber + 1,
384     NewMaster = chooseMaster(State, NewViewNumber),
385     sendToReplicas( MasterNode
386                   , Nodes
387                   , {startViewChange, NewViewNumber, MyNode}
388                   ),
389     { next_state
390     , viewChange
391     , State#{ viewNumber := NewViewNumbern
392             , masterNode := NewMaster }
393     , Timeout
394     };
```

Listing 3: Processing the VR Message Queue, Part 1 — `vr.erl`

```
397 processPrepareOrCommit( OpNumber, MasterCommitNumber, NewPrepareBuffer
398                       , #{ timeout := Timeout } = State) ->
399     AfterBuffer =
400       processBuffer( State#{ prepareBuffer := NewPrepareBuffer }
401                    , NewPrepareBuffer
402                    , MasterCommitNumber
403                    ),
404     AfterLog = processLog(AfterBuffer, MasterCommitNumber),
405     #{ masterNode := MasterNode
406      , viewNumber := ViewNumber
407      , commitNumber := CommitNumber
408      , myNode := MyNode } = AfterLog,
409     if
410         CommitNumber < MasterCommitNumber ->
411             startRecovery(State);
412         true ->
413             sendToMaster( MasterNode
414                         , {prepareOk, ViewNumber, OpNumber, MyNode}
415                         ),
416             {next_state, replica, AfterLog, Timeout}
417     end.
```

Listing 4: Processing the VR Message Queue, Part 2 — `vr.erl`

The final definition clause triggers when the replica receives a message in the current view. The replica adds the message to a sorted queue and calls a processing procedure. The processing procedure is defined separately in Listings 3 and 4.

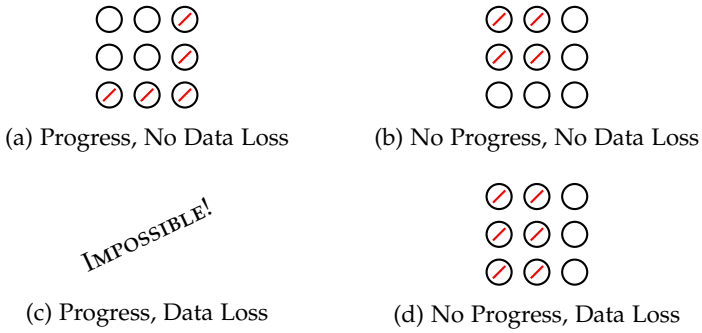
Listing 3 handles a message from an out-of-date master, i.e., the master's commit number is older than the replica's commit number. The replica first proposes a view change to the cluster. Afterwards, it changes its `gen_fsm` state to viewChange and updates the store[3] with the new view number and the new master.

Listing 4 handles messages from the current master. The prepareBuffer stores prepare messages that arrived out of order. In particular, if the log ends at operation number $n-1$ and message n is dropped by the network, all subsequent messages, $n+i$ will be buffered. The buffered messages will not be added to the log until message n is received.

The call to processBuffer moves messages to the log, if all previous messages have now been received. The call to processLog commits logged messages which have been newly committed by the master node. Lines 405–408 destructure AfterLog and bind variables for later use. The final if statement responds to the master unless the master has committed messages the replica has not yet received.[4]

[3]The store is called State in the code listings

[4]This can happen if the network drops a message whose operation number lies between the



(a) Progress, No Data Loss

(b) No Progress, No Data Loss

(c) Progress, Data Loss

(d) No Progress, Data Loss

Figure 4: Some Failed Node Distributions of PHATRAID(3,1). Each circle represent a singe VR master or replica and each column is a single PHAT cluster. The red lines indicate failures.

3 Evaluation

3.1 Resilience

We distinguish between two types of failure: cessation of progress and loss of data. In a VR cluster data loss of fully propagated, commited data only occurs when all $2f + 1$ nodes in the cluster fail [5]. In a VR cluster, progress only ceases if $f + 1$ nodes are lost. In PHATRAID the conditions for cessation of progress and loss of data are complicated by the combination of RAID and VR.

Four arrangements of node failures in PHATRAID(3,1) are depicted in Figure 4. There are three columns, each representing a PHATcluster of three nodes. The red lines indicate failures. We assume failures are independent and uniformly distributed throughout the PHATRAID group.

The top left corner, Figure 4a, depicts five of the nine nodes failing. This state is *not* a failure state. The group can still make progress and no data is lost. The two surviving clusters can reach consensus on new values and can reconstruct the failed cluster's data (by taking the XOR of the data on the two surviving clusters).

The top right corner, Figure 4b, depicts four of the nine nodes failing. This state is a failure state. The group cannot make progress; however, no data can been lost.[6] Progress is blocked because more than one cluster is unable

replica's last commit number and the master's last commit number.

[5] If more than $f + 1$ nodes die some "commited but not fully propogated data" can be lost

[6] Here and later in this paper we mean that no data which has been decided and propagated throughout the cluster can be lost. In VR and normal PHAT, by contrast, if data is decided and propagated throughout the cluster, that data can only be lost if every node is lost.

to reach consensus. Since no cluster was completely destroyed, all data still exists on at least one node.

The bottom left corner, Figure 4c, is impossible because the group cannot make progress if data has been lost.

The bottom right corner, Figure 4d, depicts six of the nine nodes failing. This state is also a failure state. The group cannot make progress and will lose data. Progress is blocked because more than one cluster is unable to reach consensus. Additionally, two complete clusters have been lost so at least half of the data cannot be recovered.[7]

3.1.1 Theoretical Analysis

In general, progress ceases if more than one cluster *cannot reach consensus* and data loss occurs if more than one cluster *completely fails*. The lower bound of failed nodes for progress cessation in PhatRaid(C,f) is:

$$2 \cdot (f + 1)$$

and the lower bound of failed nodes for data loss is:

$$2 \cdot (2f + 1)$$

The upper bound of failed nodes which still permit progress is:

$$(C - 1) \cdot f + 2f + 1$$

because $C - 1$ clusters could still make progress, e.g. if at least $f + 1$ nodes are up in each cluster, and only one cluster is completely dead.

The upper bound for no data loss is:

$$(C - 1) \cdot 2f + 2f + 1$$

because it is possible that $C - 1$ clusters have at least one node up and only one cluster has no nodes up.

The bounds presented above cannot fully describe the failure threshold of a PhatRaid cluster because of the interplay between VR and RAID. Instead, we discuss the probability of group failure given n node failures.

[7] If only the cluster with the parity bit remains then all useable data has been lost

Group failure is dependent on the arrangement of failed nodes. For example, Figure 4b depicts a situation wherein only four failures stops progress. In contrast, Figure 4a depicts a situation wherein five failures does *not* stop progress.

In a PhatRaid(3,1) group, less than one-quarter of four-failure arrangements stop progress. The necessary condition for progress cessation is that two or more clusters have lost at least $f + 1$ nodes. Only the two-in-two, four-failure arrangement stops progress. We can count the number of such arrangements:

$$\binom{3}{2}\binom{3}{2}^2 = 3^3 = 27$$

The first term represents choosing the two clusters of three that will fail. The second term represents choosing which two nodes in each cluster of three nodes will fail.

The total number of four-failure arrangements is simply:

$$\binom{9}{4} = 126$$

If all nodes fail uniformly and independently, then the probability of four nodes causing PhatRaid group failure is

$$\frac{27}{126} \approx 0.21$$

When five nodes fail, they can be distributed into each cluster in three different ways: (3,1,1), (3,2,0) and (2,2,1). The first case does not stop progress because only one cluster fails. The later two cases stop progress because two clusters fail. We calculate the probability of PhatRaid group failure when five nodes fail:

$$
\begin{aligned}
(3,1,1) \quad & \binom{3}{1}\binom{3}{3}\binom{3}{1}\binom{3}{1} = 3^3 = 27 \\
(3,2,0) \quad & \binom{3}{1}\binom{2}{1}\binom{3}{3}\binom{3}{2} = 2 \cdot 3^2 = 18 \\
(2,2,1) \quad & \binom{3}{1}\binom{3}{2}\binom{3}{1}\binom{3}{1} = 3^4 = 81 \\
\text{5-Failure Arrs.} \quad & \binom{9}{5} = 126 \\
\text{Failure Prob.} \quad & \Pr\left[\text{group failure} \,|\, \textsc{PhatRaid}(3,1),\, n_{failed} = 5\right] \\
& = \frac{81+18}{126} = \frac{99}{126} \approx 0.79
\end{aligned}
$$

A VR or PAXOS cluster of size nine is resilient to $f = 4$ failures. In contrast, PHATRAID(3,1) is absolutely resilient to $f = 3$ failures, has high probability of resilience to $f = 4$ failures, and has low probability of resilience to $f = 5$ failures.

The general formula for the failure probability distribution is left as an exercise for the reader, but it is clear that a PHATRAID(C,f) group is absolutely resilient to $2(f+1)-1$ or fewer failures. In addition, a PHATRAID(C,f) group is likely resilient to $2(f+1)$ failures and unlikely resistant to $(C-1)\cdot f + 2f + 1$ failures.

For exactly $2(f+1)$ node failures, the probability of a PHATRAID(C,f) group failure is:

$$
\Pr\left[\text{group failure} \,|\, \textsc{PhatRaid}(C,f),\, n_{failure} = 2(f+1)\right] = \frac{\binom{C}{2}\binom{2f+1}{f+1}^2}{\binom{C\cdot(2f+1)}{2(f+1)}}
$$

For example, for a PHATRAID(6,1) we calculate:

$$
P\left[\text{group failure} \,|\, \textsc{PhatRaid}(6,1),\, n_{failure} = 4\right] = \frac{135}{3000} \approx 0.05
$$

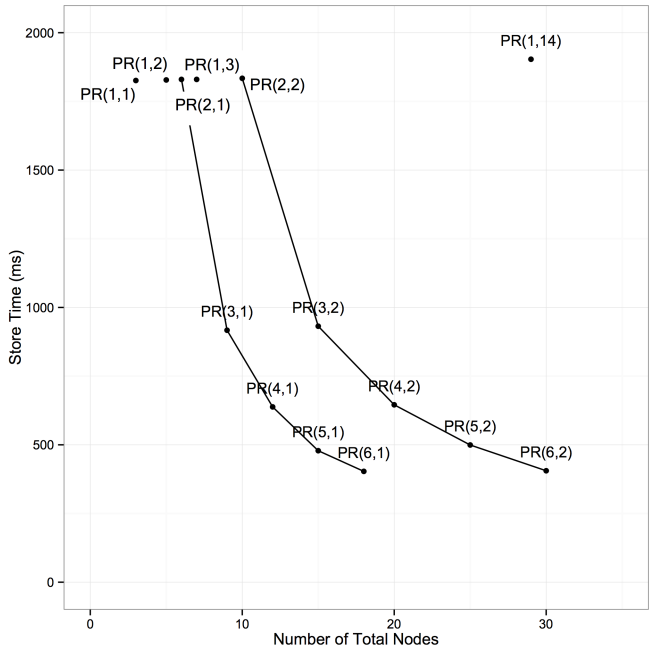


Figure 5: Scatter Plot of Various PhatRaid(C,f) Configurations

3.2 Performance

We evaluated a prototype PhatRaid(C,f) implementation. We simulated disk latency with sleeps. All experiments are run on a 2013-model MacBook Pro with 8GB of RAM, and a 3GHz Intel Core i7. We found that latency scales asp $\frac{1}{C-1}$. This is the expected behavior for breaking a file into $C-1$ chunks. We always use one bit of party.

Figure 5 is a scatter plot of the time to store a 100 MB file in a PhatRaid group. Each point represents a different configuration. The x-axis indicates the number of nodes needed for that configuration. The x-axis can also be seen as a measure of cost to create the configuration.

The slow linear growth from PhatRaid(1,1) to PhatRaid(1,14) reveals the cost of communication in large Phat clusters. This phenomenon is also present in the slight increase in cost from PhatRaid(C,1) to PhatRaid(C,2).

As expected PhatRaid(2,f) does not perform better than non-RAID Phat (i.e. PhatRaid(1,f)) because the data is not sharded into pieces, and the parity bit acts as a complete duplicate of the file.

4 Future Work

We would like to consider adding more than one parity bit. For some applications, the diminishing returns on latency are not worth the loss of failures. In that case, we would do well to add more parity bits, thus allowing more clusters to completely fail.

We would like to perform a real evaluation rather than a simulation. It seems that as file size increases, PhatRaid becomes more appealing for lower commit latency. It is not clear from our simulation, however, when the file size is small enough for other effects (such as network latency, communications overhead) to become more important.

5 Conclusion

We've presented RAID on a distributed file system which shows that sharding files can improve latency for large files while maintaining reasonable failure probabilities.

References

[1] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[2] David Johnson. The advantages and disadvantages of solid state drives. `http://www.champsbi.com/the-advantages-and-disadvantages-of-solid-state-drives/`, March 2014.

[3] Eddie Kohler. Harvard computer science 260r. `http://read-new.seas.harvard.edu/cs260r/2014/w/Phat`, 2014.

[4] Barbara Liskov and James Cowling. Viewstamped replication revisited. *MIT Technical Report*, 2012.

[5] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.