Storage services

Badri Nath
Rutgers University
badri@cs.rutgers.edu

Storage Virtualization

- Typically Storage ↔ drive
- C:/ D:/ F:/ ...
- What if we wanted one virtual drive?
- Create, read, write on V:/ [1] … [N]
- Centralized:
  - Create a MAP from V:/ to Physical Drive
  - Mapping scheme

Average price of storage

Storage Virtualization

- What happens when a physical drive fails?
- If Vblocks are not replicated, then cannot access Vblocks
- Make redundant copies on different drives
- Design a replication scheme
Storage Virtualization

- Distributed
- Physical drives are attached to servers connected by a network
- Issues?
- Metadata, failures, scalability, concurrency, consistency, backup, reconfiguration….

Petal: Distributed Virtual Disks

- A distributed storage system that provides a virtual disk abstraction separate from the physical resource
- The virtual disk is globally accessible to all Petal clients on the network
- Virtual disks are implemented on a cluster of servers that cooperate to manage a pool of physical disks
- Advantages
  - recover from any single failure
  - transparent reconfiguration and expandability
  - load and capacity balancing
  - low-level service (lower than a DFS) that handles distribution problems

Logical System View

Physical system view
Virtual to Physical Translation

Global State Management
- Global state such as current members and current vdisks is replicated across servers and consistently maintained
- Based on Leslie Lamport’s Paxos algorithm.
- A majority is needed to update global state.
- Any server can be added/removed in the presence of failed servers.

Global State Management

The Consensus Problem
- A collection of processes can propose values
- Only a single of the proposed values must be chosen
- Model Assumptions
  - Distributed architecture (no centralized arbiter, no global clock)
  - Asynchronous system (arbitrary message delays)
  - Fail stop failures

The Consensus Problem: paxos algorithm
- Three classes of agents:
  - Proposers (propose the values)
  - Acceptors (accept the values)
  - Learners (learn the current chosen value)
- Propose some value (new entry in the log, new server in a table – as in Petal, a decision value-go or no go)
- After execution of the algorithm, everyone will agree on a value
Solution 1:
- We need to agree on one value
- Many acceptors
- Majority wins
- A value is chosen if the majority of acceptors accept it
- Lets assume the acceptors accept the first proposal
- If there are multiple proposers, then we end up with different accepted values — no consensus
- We need to ensure that all accepted proposals have the same value

**Paxos algorithm; Phase 1**
- P1: Prepare request
  1) A proposer chooses a new proposal with number and possible value \((n_{\text{Proposal Number, NodeID}}, v)\) > and sends it to all acceptors
  2) If an acceptor receives a prepare request with a proposal number \(n >\) any prepare request it has already responded to, then it acks (ack, \(n, n', v\)) or (ack, \(n\))
  - Promise not to accept any proposal \(< n\)
  - May suggest a value \(v\) of the highest-number proposal it has accepted so far

<table>
<thead>
<tr>
<th>number</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>(v)</td>
</tr>
</tbody>
</table>

**Paxos algorithm; Phase 2**
- P2: Accept request
  1) If the proposer receives responses from a majority of the acceptors, then it can issue an accept request \((n, v)\) where \(n\) is the proposal number in the prepare request and value \(v\) is of the highest-number proposal
  2) If an acceptor receives an accept request \((n, v)\), it accepts the proposal unless it has responded to a prepare request having a number greater than \(n\)

<table>
<thead>
<tr>
<th>number</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>(v)</td>
</tr>
</tbody>
</table>

**Learning The “Right” Value for Proposal**
- A learner can ask the latest value by asking any participant
- Acceptors can send decided value to all learners
- A Learner after receiving \((n, v)\) from a majority of acceptors can send \(V\) to other learners
Applying Paxos to Petal

- Need to agree on current members, vdisks
- Run paxos when
  - Add or remove servers
  - Add or remove vdisks
  - After reboot

Virtual to Physical Translation

- \(<\text{virtual disk}, \text{virtual offset}> \rightarrow <\text{server, physical disk}, \text{physical offset}>\)
- Three data structures: virtual disk directory, global map, and physical map
- The virtual disk directory and global map are globally replicated and kept consistent
- Physical map is local to each server
- One level of indirection (virtual disk to global map) is necessary to allow transparent reconfiguration.

Virtual to Physical Translation (cont’d)

- The virtual disk directory translates the virtual disk identifier into a global map identifier
- The global map determines the server responsible for translating the given offset (a virtual disk may be spread over multiple physical disks). The global map also specifies the redundancy scheme for the virtual disk
- The physical map at specific server translates global map identifier and the offset to a physical disk and an offset within that disk. Physical map is similar to a page table. The physical map at a server pertains to disks of that server

Virtual to Physical Translation

- VDir- virtual directory maps virtual disks \(\rightarrow <\text{GMap, epoch-number}>\)
- GMap – maps offset \(\rightarrow\) redirect server /*immutable for one epoch */
- Pmap per server, maps \(<\text{GmapID, epoch, offset}> \rightarrow \) physical block, translate 64 K at a time
**Support for Backup**

- Petal simplifies a client’s backup procedure by providing a snapshot mechanism.
- Petal generates snapshots of virtual disks using copy-on-write. Creating a snapshot requires pausing the client’s application to guarantee consistency.
- A snapshot is a virtual disk that cannot be modified.
- Snapshots require a modification to the translation scheme. The virtual disk directory translates a virtual disk id into a pair <global map id, epoch #> where epoch # is incremented at each snapshot.
- At each snapshot a new tuple with a new epoch is created in the virtual disk directory. The snapshot takes the old epoch #.
- All accesses to the virtual disk are made using the new epoch #, so that any write to the original disk create new entries in the new epoch rather than overwrite the blocks in the snapshot.

**Virtual Disk Reconfiguration**

- Needed when a new server is added or the redundancy scheme is changed.
- Steps to perform it at once (not incrementally) and in the absence of any other activity:
  1. create a new global map with desired redundancy scheme and server mapping.
  2. change all virtual disk directories to point to the new global map.
  3. redistribute data to the servers according to the translation specified in the new global map.
- The challenge is to perform it incrementally and concurrently with normal client requests.

**Incremental Reconfiguration**

- First two steps as before; step 3 done in background starting with the translations in the most recent epoch that have not yet been moved.
- Old global map is used to perform read translations which are not found in the new global map.
- A write request only accesses the new global map to avoid consistency problems.
- Limitation: the mapping of the entire virtual disk must be changed before any data is moved. -> lots of new global map misses on reads -> high traffic. Solution: relocate only a portion of the virtual disk at a time. Read requests for portion of virtual disk being relocated cause misses, but not requests to other areas.

**Redundancy with Chained Data Placement**

- Petal uses chained-declustering data placement.
- Two copies of each data block are stored on neighboring servers.
- Every pair of neighboring servers has data blocks in common.
- If server 1 fails, servers 0 and 2 will share server’s read load (not server 3).
Chained Declustering

Chained Data Placement (cont’d)

- In case of failure, each server can offload some of its original read load to the next/previous server. Offloading can be cascaded across servers to uniformly balance load.
- Advantage: with a simple mirrored redundancy, the failure of a server would result in a 100% load increase to another server.
- Disadvantage: less reliable than simple mirroring - if a server fails, the failure of either one of its two neighbor servers will result in data becoming unavailable.
- In Petal, one copy is called primary, the other secondary.
- Read requests can be serviced by any of the two servers, while write requests must always try the primary first to prevent deadlock (blocks are locked before reading or writing, but writes require access to both servers).

Read Request

- The Petal client tries the primary or secondary server depending on which one has the shorter queue length. (Each client maintains a small amount of high-level mapping information that is used to route requests to the "most appropriate" servers. If a request is sent to an inappropriate server, the server returns an error code, causing the client to update its hints and retry the request)
- The server that receives the request attempts to read the requested data.
- If not successful, the client tries the other server.

Write Request

- The Petal client tries the primary server first.
- The primary server marks data busy and sends the request to its local copy and the secondary copy.
- When both complete, the busy bit is cleared and the operation is acknowledged to the client.
- If not successful, the client tries the secondary server.
- If the secondary server detects that the primary server is down, it marks the data element as stale on stable storage before writing to its local disk.
- When the primary server comes up, the primary server has to bring all data marked stale up-to-date during recovery.
- Similar if secondary server is down.
Petal Performance - Latency

Table 1: Latency of a Chained-Declarative Virtual Disk

<table>
<thead>
<tr>
<th>Request</th>
<th>Local Disk</th>
<th>KZ20 Log</th>
<th>NVRAM Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 byte Read</td>
<td>9</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8 Kbyte Read</td>
<td>13</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>64 Kbyte Read</td>
<td>31</td>
<td>78</td>
<td>78</td>
</tr>
<tr>
<td>512 byte Write</td>
<td>10</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>8 Kbyte Write</td>
<td>12</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>64 Kbyte Write</td>
<td>20</td>
<td>40</td>
<td>35</td>
</tr>
</tbody>
</table>

Single client generates requests to random disk offsets

Petal Performance - Throughput

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Throughput</th>
<th>Failed Throughput</th>
<th>% of Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 byte Read</td>
<td>3.50 Mbps</td>
<td>2.50 Mbps</td>
<td>75%</td>
</tr>
<tr>
<td>8 Kbyte Read</td>
<td>43.1 Mbps</td>
<td>14.6 Mbps</td>
<td>33%</td>
</tr>
<tr>
<td>64 Kbyte Read</td>
<td>6.6 Mbps</td>
<td>6.6 Mbps</td>
<td>100%</td>
</tr>
</tbody>
</table>

Failed configuration = one of 4 servers has crashed

Frangipani

- Petal provides disk interface -> need a file system
- Frangipani is a file system designed to take full advantage of Petal
- Frangipani's main characteristics:
  - All users are given a consistent view of the same set of files
  - Servers can be added without changing configuration of existing servers or interrupting their operation
  - Tolerates and recovers from machine, network, and disk failures
  - Very simple internally: a set of cooperating machines that use a common store and synchronize access to that store with locks
  - Easy to administer
  - Exhibits good performance, scaling, and load balancing
Frangipani

- Petal takes much of the complexity out of Frangipani
  - Petal provides highly available storage that can scale in throughput and capacity
- However, Frangipani improves on Petal, since:
  - Petal has no provision for sharing the storage among multiple clients
  - Applications use a file-based interface rather than the disk-like interface provided by Petal
- Problems with Frangipani on top of Petal:
  - Some logging occurs twice (once in Frangipani and once in Petal)
  - Cannot use disk location in placing data, cause Petal virtualizes disks
  - Frangipani locks entire files and directories as opposed to individual blocks

Frangipani Structure

Components of Frangipani

- File system core
  - implements the Digital Unix vnode interface
  - uses the Digital Unix Unified Buffer Cache
  - exploits Petal’s large virtual space
- Locks with lease
- Write-ahead redo log
Frangipani: Disk Layout

- A Frangipani file system uses only 1 Petal virtual disk
- Petal provides a $2^{64}$ bytes of “virtual” disk space
  - Commits real disk space when actually used (written)
- Frangipani breaks disk into regions
  - 1st region stores configuration parameters and housekeeping info
  - 2nd region stores logs — each Frangipani server uses a portion of this region for its log. Can have up to 256 logs.
  - 3rd region holds allocation bitmaps, describing which blocks in the remaining regions are free. Each server locks a different portion.
  - 4th region holds inodes
  - 5th region holds small data blocks (4 Kbytes each)
  - Remainder of Petal disk holds large data blocks (1 Tbyte each)
  - 16 M large files

Frangipani: File Structure

- First 16 blocks (64 KB) of a file are stored in small blocks
- If file becomes larger, store the rest in a 1 TB large block

Logging

- Frangipani uses a write ahead redo log for metadata
  - log records are kept on Petal
  - In-place update done after log entry
- Data is written to Petal
  - on sync, fsync, or every 30 seconds
  - on lock revocation or when the log wraps
- Each machine has a separate log (128 KB)
  - Circular buffer with periodic reclaiming
  - Two distinct disks
  - reduces content, independent recovery
**Frangipani: Dealing with Failures**

- If a server crashes, the system detects the failure and another server uses the log to recover.
  - Because the log is on Petal, any server can get to it.

**Frangipani: Synchronization & Coherence**

- Frangipani has a lock for each log segment, allocation bitmap segment, and each file.
- Multiple-reader/single-writer locks. In case of conflicting requests, the owner of the lock is asked to release or downgrade it to remove the conflict.
- A read lock allows a server to read data from disk and cache it. If a server is asked to release its read lock, it must invalidate the cache entry before complying.
- A write lock allows a server to read or write data and cache it. If a server is asked to release its write lock, it must write dirty data to disk and invalidate the cache entry before complying. If a server is asked to downgrade the lock, it must write dirty data to disk before complying.
  - Invalidate write cache, if releasing the lock.

**Frangipani: Lock Service**

- Fully distributed lock service for fault tolerance and scalability.
- Lock requestors are frangipani servers.
- How to release locks owned by a failed Frangipani server?
  - The failure of a server is discovered when its "lease" expires. A lease is obtained by the server when it first contacts the lock service. All locks acquired are associated with the lease. Each lease has an expiration time (30 seconds) after its creation or last renewal. A server must renew its lease before it expires.
  - When a server fails, the locks that it owns cannot be released until its log is processed and any pending updates are written to Petal.

**Locks**

- Locks are moderately coarse-grained.
  - Protects entire file or directory.
- Locks are partitioned into lock groups (100).
- Use consensus algorithm (paxos) to agree on lock group assignment among servers.
- On crash, Retreive lost state from clerks.
**backup**

- How to backup a Frangipani file system?
- Petal offers snapshots. Exploit its feature
- Copy snapshots to tertiary storage
- Restore entire Petal snapshot (including logs), and do crash recovery
- What about recovering individual files? Not very efficient, search all logs

**Frangipani: Performance**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
<th>Adfs</th>
<th>Frangipani</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create Directories</td>
<td>0.69</td>
<td>0.66</td>
</tr>
<tr>
<td>2</td>
<td>Copy Files</td>
<td>4.3</td>
<td>4.3</td>
</tr>
<tr>
<td>3</td>
<td>Delete Files</td>
<td>4.7</td>
<td>4.4</td>
</tr>
<tr>
<td>4</td>
<td>Remove Files</td>
<td>4.8</td>
<td>4.3</td>
</tr>
<tr>
<td>5</td>
<td>Compile</td>
<td>22.8</td>
<td>22.7</td>
</tr>
</tbody>
</table>

Table 1: Modified Andrew Benchmark with unmount operations. We compare the performance of two file system configurations: local access (with and without NVRAM) in the DIGITAL Unix Advanced File System (Adfs), Frangipani, and Frangipani with an NVRAM buffer added between Petal and the disks. We unmount the file system at the end of each phase. Each table entry is an average elapsed time in seconds; smaller numbers are better.

**Frangipani: Scalability**

Figure 5: Frangipani Scaling on Modified Andrew Benchmark. Several Frangipani servers simultaneously ran the Modified Andrew Benchmark on independent node sets. The y-axis gives the average elapsed time (in seconds) to complete the benchmark.
**Frangipani: Scalability**

![Graph](image1)

**Figure 6:** Frangipani Scaling on Uncached Read. Several Frangipani servers simultaneously read the same set of files. The dotted line shows the linear speedup curve for comparison.

![Graph](image2)

**Figure 7:** Frangipani Scaling on Write. Each Frangipani server writes a large private file. The dotted line shows the linear speedup curve for comparison. Performance tapers off early because the ATM links to the main servers become saturated.