Petal and Frangipani

Questions from Reviews

- What is the motivation for Petal? Why not use a file system?
  - What semantic information is lost?
  - What is gained?
  - Does file locality matter when you have a multi-programmed workload and 100 disks working in parallel?
  - Must semantic information be lost?
- Who re-replicated data for a permanently failed node?
- Chained declustering leads to little spread of data
- Many reviews written as if you had not read Frangipani…

Petal paper notes

- This is an ASPLOS paper – limited to 10 pages, so not as detailed as other systems papers
- Written at a time when systems were smaller, machines more expensive, so prototyping on a large network may not be feasible
- Performance compared to local disk:
  - They use a fast network, so network overhead small
  - They have multiple disks, multiple disk heads, so can have better locality for small workloads
  - Performance for Andrew benchmark bogus – it is a bogus benchmark

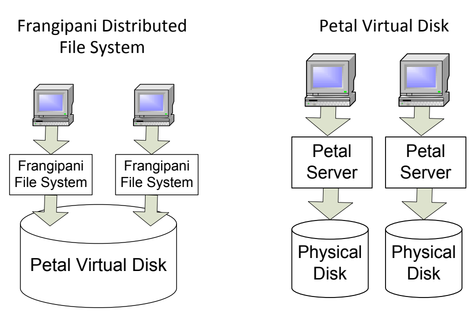
Petal

- Virtual disk
  - Looks like a large block device to clients
  - Good for use by a cluster to share storage
- Why?
  - Scalable, elastic storage for single nodes.
    - E.g. our mail server, SMB or NFS server can grow to store any amount of data, can
    - Works with legacy node-local file systems
- How big?
  - Small cluster (dozen machines?
  - High-quality machines so permanent failures rare

Overall Points

- Layered approach: separate block allocation & placement from file system naming and objects
- Uses large-sparse address space to represent logical objects
- Shared disks+locks only, no other coordination

Layered Approach



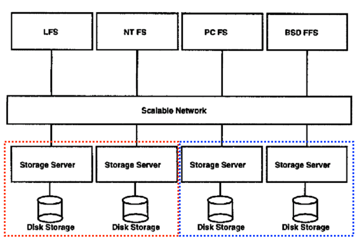
Petal: Distributed Virtual Disks

- A distributed storage system that provides a virtual disk abstraction separate from the physical resource
  - Note: current SAN products did not exist yet
- The virtual disk is globally accessible to all Petal clients on the network (ATM)
  - Note: current choice (fibrechannel) did not exist
- 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
- QUESTION: What is benefit of block layer?
  - Simpler semantics (sharing a block is easier to reason about that simultaneous update to files or directories)
  - Reliable block layer simplifies upper level; most local file systems assume it to be true (or else fail catastrophically)

Petal

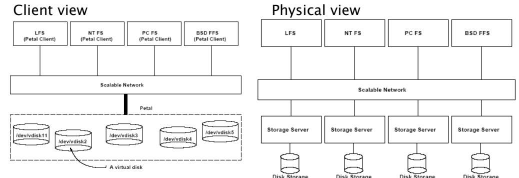
- A Distributed Virtual Disk
  - No files
  - No synchronization
  - No meta-data

Petal



PETAL: Distributed Virtual Disks

Client view Physical view

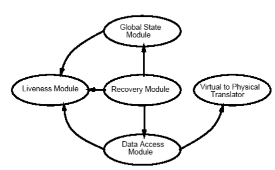


In which context this might not be the best approach?

Design choice

- Petal config data stored within petal, to reduce state at client
  - Other papers say you should do more work at client for scalability
- QUESTION: Is there a conflict?
  - What is the work involved? sending requests to the right nodes (Coda style – client sends twice)
  - Petal provides strict consistency, so benefits from server doing work

PETAL: Design



- Servers maintain most of the state.
- Clients maintain only 'hints'
- Algorithm guarantees recovery from random failures of servers and network connectivity as long as majority of servers are up and communicated.

Interface Abstraction

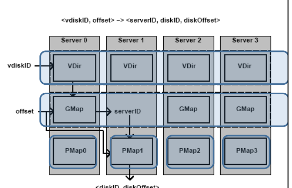
- Virtual disks:
  - Allow multiple file systems to simultaneously use Petal
  - Consistency is within a virtual disk
- Virtual disk exports virtual block addresses
  - Allocates backing space on demand as address space is used
- Now called a Storage Area Network (SAN)
  - Used in our department by mail server, NFS servers, to flexibly deploy storage where needed
- Made possible by fast networks (< 1ms) so RTT fast compared to disk

Virtual to Physical Translation

- <virtual disk, virtual offset> -> <server, physical disk, 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.

PETA: Virtual → Physical

- Virtual ID -> Global ID
- Global Map identifies correct server.
  * partitions block address space
- Physical Map in the server translates the GID into the Physical disk and real offset.



Virtual to Physical Translation (cont'd)

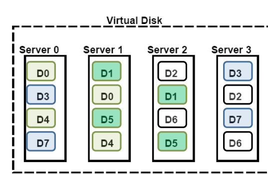
- The virtual disk directory translates the virtual disk identifier into a global map identifier
- The global map determines the server responsible for translating the given offset
  - Partitioning: a virtual disk may be spread over multiple physical disks).
  - The global map also specifies the redundancy scheme for the virtual disk
    - e.g. how many copies
  - NOTE: GMAP doesn't change, instead create new GMAP + new mapping of virtual disk to GMAP
- 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
  - QUESTION: Compare to Dynamo?
- Global map of keys to buckets is replicated, map of keys to disk space is local
  - QUESTION: Compare to coda?
    - Global map of volumes to servers replicated,
    - map of files to disk blocks is local

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

PETAL: Failure tolerance

- Chained-declustered data access.
  - 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)
- Stores data twice
  - QUESTION: is that enough?
    - Compare to RAID – one extra disk
    - Mirroring – one extra disk
    - Supports backup



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

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 (paxos)
  3. redistribute data to the severs 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
  - May want to use consistent hashing to avoid data movement
- 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

Read Request

- The Petal client tries 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
  - Locks data during read (because granularity may be higher than device granularity, e.g. 64kb block vs 512b sector)
- 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** (e.g. locked)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
- KEY ELEMENT: primary/backup note persistently whenever they modify data without replication, so it can be reconciled later

Petal points

- Multi-level mapping
  - global mapping coarse, widely replicated, immutable
  - Local mapping simple
- primary-backup replication (2 nodes)
  - mark data no replicated for reintegration
- chained-declustering for load balancing after failure
- Export multiple virtual disks+snapshots
- NOTE: can be popular for virtual machines to support virtual disks (no sharing)

FRANGIPANI: Motivation

- Why a distributed file system?
  - Scalability, Performance, Reliability, Durability
- Frangipani
  - Scalability → Easy to add more components
    - Shared disks in petal, front end frangipani servers
  - Administration → Simple
    - No manual assignment of users/files to servers
    - Consistent backups
  - Tolerates and recover from machine, network, and disk failures.
    - Without operator intervention
    - E.g. no manual conflict resolution

Frangipani points

- Only shared state is on disk or in locks
  - "shared nothing" design
- Partition large, sparse address space for simplicity
  - private info (bitmaps, logs)
  - Metadata
  - small file data
  - large file data
- Remote recovery from failure
  - logs stored on shared disks
- Locks for strict consistency
  - must integrate into recovery

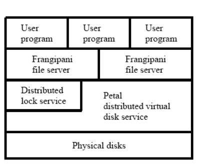
FRANGIPANI: Overview

- Focused on Clustered systems
- Simple design
  - Many assumptions and constraints → <u>Performance impact?</u>
    - Trusted environment: Limited security considerations
    - Lack of portability: Runs in kernel level
    - Two layered design: PETAL is used
      - ⇓ control : ⇑ simplicity

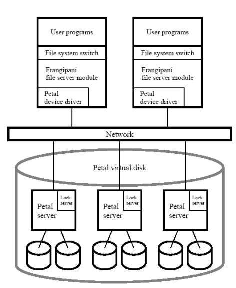
Security and Administration

- There's a lot of trust here
  - Frangipani servers trust each other
  - Frangipani servers trust petal
  - Frangipani servers trust the locking service
- Single Administrative Structure
  - requires secure communication
- Some helpful hints at security provided, but largely incidental and speculative

FRANGIPANI: Architecture overview



Frangipani layering



One possible configuration

Frangipani

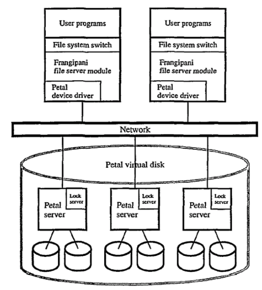
- Petal takes much of the complexity out of Frangipani
  - Petal provides highly available storage that can scale in throughput and capacity
- However, Frangipani improves on Petal, since:
  - Petal has no provision for sharing the storage among multiple clients
  - Applications use a file-based interface 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
- QUESTION: How important is it for the file system to place data?
  - QUESTION: What does it really need to know? What things are near what other things
  - In a large system with multiple servers & multiple disks, is this relevant?
  - Petal focuses more on throughput (lots of bandwidth) than getting latency down
  - Large files can be striped

Frangipani: File Structure

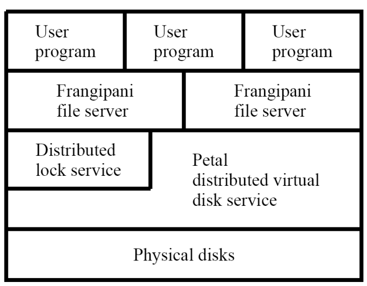
- First 16 blocks (64 KB) of a file are stored in small blocks
- If file becomes larger, store the rest in a 1 TB large block

Frangipani Structure

Note: untrusted clients can be served by running NFS/CIFS on a trusted server
NOTE: we use this architecture in our department



Frangipani System Structure



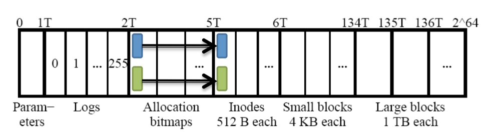
Frangipani: Disk Layout

- 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)
- QUESTION: Why? What is the benefit?
  - Not worry about complex block allocation policies
  - Rely on virtual addresses to make things private (e.g. logs) or describe what synchronization is needed (e.g. bitmaps, file data)

Locality

- Does building on Petal sacrifice locality?
  - Files are layed out contiguously in virtual address space (if they are > 64 kb)
  - Petal can lay out blocks to optimize sequential performance
    - Strip chunks across machines
- What kind of locality?
  - Files within a directory?
  - Temporal locality via caching

FRANGIPANI: Disk Layout



2^{64} Byte address space limit chosen based on usage experience.

<u>Separating inodes and data blocks completely might result in performance hit.</u>

FRANGIPANI: Logging and Recovery

- Each Frangipani server has its own log in PETAL.
  - Log records have increasing LSN (to find tail)
- Logs bounded in size → stored in a circular buffer → when full, system reclaims oldest 25%.
- Used for standard journaling:
  - Log metadata before changing metadata blocks
  - Logs store only metadata → Speeds up
- On recovery:
  - Changes applied only if record version > block version
  - Change protected by locks must be written to real location before releasing lock
    - Reading data must produce correct results.
- Metadata blocks are versioned
  - Metadata blocks are reused only by new metadata blocks:
  - Data block do not have space reserved for version numbers

Logging and Fault Tolerance

- Only Meta-Data is logged
- Logging is stored on the petal virtual disk
- Any server on network can do recovery
  - Can take over another log & replay it
- Petal must maintain physical access to each block of user data
  - Data is replicated by petal

Frangipani: Dealing with Failures

- Write-ahead redo logging of metadata; user data is not logged
- Each Frangipani server has its own private log
- Only after a log record is written to Petal does the server modify the actual metadata in its permanent locations
- 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.

Synchronization

- Read / Write locks
- Applies to files, logs, and bitmap segments
- Global ordering for deadlock prevention
- Provided by independent server or cluster

FRANGIPANI: Cache

- Dirty data is flushed to disk when downgrading locks: Write → Read
- Cache Data is invalidated if it is releasing the lock: Read → No Lock → Someone requested a Write Lock.
  - Dirty data is not sent to the new lock owner.
  - Frangipani servers communicate only with Petal.
- One lock protects inode and data blocks
  - Per-file granularity.

Locks

- Servers can be requested to release or downgrade locks
  - Doing so requires flushing and/or cache invalidating
  - Otherwise, they'll just hold locks
- 30 second expiration time
- Local lock management module: Clerk
  - It opens a table of locks, tracks which ones are held, etc.

Frangipani: Synchronization & Coherence

- Frangipani has a lock for each log segment, allocation bitmap segment, and each file
  - QUESTION: Compare this to AFS – similar granularity of locking
- 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 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

Frangipani: Lock Service

- Fully distributed lock service for fault tolerance and scalability
- How to release locks owned by a failed Frangipani server?
  - A lease is obtained by the server when it first contacts the lock service.
    - The failure of a server is discovered when its "lease" expires.
    - 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

Frangipani: Lock Service

- Lock Server → Multiple Read/ Single Write locks
- <u>2 servers might try to write same file, both will keep acquiring and releasing locks</u>
- Asynchronous messages: request, grant, revoke, release. (Optional Synchronous)
- Global lock state replicated with Paxos
  - List of servers
  - Partitioning of locks
  - List of clients accessing locks
- Crash of Lock Servers
  - Same as Petal → heartbeats between servers → majority consensus to tolerate network partitions.
  - If lock server crashes, locks managed by it are redistributed.
    - Lock state is retrieved from the clerks
  - If frangipani server fails, the locks are assigned to another Frangipani server and a recovery process is executed.

Backups

- Leverages petals snapshot feature
- Includes logs, so we can use it to restore

Performance

- Beneficial Parallelism
  - Multiple disks (striped)
  - Multiple servers (load balancing)
- Advanced Unix File System gives good performance through striped local disks
- Performance was pretty good (vs AdvFS)

Scalability

- Claim: Scales up to the limits imposed by the network
- Reality: Testing in this version was limited
  - Future tests showed…
- Andrew Benchmark
  - Reads <u>might</u> be linear
  - Writes are bounded by network
- Lock contention doesn't scale with read-ahead

Performance

- Where logging generally makes a system slow, Frangipani uses the logging function in order to improve its performance (Section 4). Would you clarify the mechanism enabling the performance improvement?
  performance improvement through asynchronous meta data updates and avoiding using fsck to check for metadata problems during recovery
- What happens if two or more server try to write to the same file? It seems this could cause performance issues because system call could cause a lock revocation request with every write and this can result in the lock holder to flush the dirty data to the Petal.
  paper argument – frangipani for engineering workloads – concurrent write access to single file rare – finer granularity locking can be implemented for other work loads
- Separating inodes and data blocks completely into different regions on petal might result in the inodes and data blocks to be on different servers within petal. How much does this issue affect the cost of file lookups?
  depends on characteristics of Petal virtual disk - benchmarks do show performance hit for reads

Two layered approach – pros and cons

- Can files migrate from one disk to another so that they will be closer to the machine that is using them (like objects in a distributed object system)? This might especially make sense if user programs are running on the same machines as the Petal servers.
  maybe logs/data associated with a frangipani server can be stored on closest petal server – file migration? Need knowledge of how files are stored – two layered approach makes this difficult
- Suppose that a user wants their machine to become a Petal server. Can their existing file system be made accessible via Frangipani, or would they have to copy their data into Frangipani and wipe (part of) their disk?
  frangipani = new file system = format disk ☺ (can't say what petal would do with user's disk)
- I believe the idea of having a file system composed of multiple layers (e.g. distribution/replication layer and lock management layer) allows for both component reusability (e.g. many different file systems could be atop Petal) and simpler implementations (The paper claims Frangipani took 2 months to write). Do you think performance could have been higher if Frangipani had been built "inside" Petal?
  abstraction/modularization/simplicity vs. performance/more control

vs. Centralized File Server

- How does a network file system differ from something like a centralized file server in terms of performance and availability?
- Will it be easier to implement security and access control in a file server?

performance – parallel multiple access vs. central access point – access point might be bottleneck
availability – centralized file server – low
security/access control – should be easier in centralized file server

Recovery

- In section 4 they describe file recovery will offer the same guarantees as a Unix file system but is this true? Is it safe to make this kind of assumption in a distributed world?
  provided logs are accessible and there is some frangipani to run them
- When a server fails it can be restarted with an empty metadata log. Would you say that this could be a problem?
  Log stored on petal – another frangipani server runs the logs

Implementation

- The abstraction of the virtual disk is used here to make multiple physical disks appear as one disk to the client. However, do you think this abstraction could be useful in other contexts? Notice that the abstraction is devoid of assumptions of how the underlying storage devices are designed (no dependence on cylinders, rotation, etc.)
  Context here –cluster file system - other contexts?
- The paper describes a scalable distributed file system, while scalability is important to the company providing web service using a lot of servers (ex. Google). How are the two kinds of scalability different or related to?
  Google servers scalability - mostly for scaling crawl data size
  Frangipani – petal servers to scale size of virtual disk, frangipani and lock servers to scale number of users

---

- Regarding the recovery of Frangipani servers, the paper mentions that it requires a clerk of another Frangipani machine to perform system recovery and process all the server's logs, and that recovery server is itself 'protected' by a lease on the lock it has of the failed server so that if anything happened to it, there will be a recovery recovery server. Now what happens if the lock server dies at the same time?
  Problem- lock server crash handling depends on clerk for lock state – frangipani server (clerk) crash handling requires lock server to give lock to another frangipani server (clerk)
- Does avoiding cache-to-cache transfer give any explicit advantage other than simplicity? In doing so, what is sacrificed here? Does it matter at all?
  performance sacrificed for simplicity – dirty cache written to disk and read from disk rather than directly from cache – logging complications otherwise

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
- QUESTION: How important is it for the file system to place data?
  - QUESTION: What does it really need to know? What things are near what other things
  - In a large system with multiple servers & multiple disks, is this relevant?
  - Petal focuses more on throughput (lots of bandwidth) than getting latency down
  - Large files can be striped