FAQ for Frangipani, Thekkath, Mann, Lee, SOSP 1997

Q: Why are we reading this paper?

A: Primarily as an illustration of cache coherence.

But there are other interesting aspects. The idea of each client having its own log, stored in a public place so that anyone can recover from it, is clever. Further, the logs are intertwined in an unusual way: the updates to a given object may be spread over many logs. This makes replaying a single log tricky (hence Frangipani's version numbers). Building a system out of simple shared storage (Petal) and smart but decentralized participants is interesting, particularly recovery from the crash of one participant.

Q: How does Frangipani differ from GFS?

A: A big architectural difference is that GFS has most of the file system logic in the servers, while Frangipani distributes the logic over the workstations that run Frangipani. That is, Frangipani doesn't really have a notion of file server in the way that GFS does, only file clients. Frangipani puts the file system logic in the clients in order to allow them to perform file system operations purely in their local caches. This makes sense when most activity is client workstations reading and writing a single user's (cached) files. Frangipani has a lot of mechanism to ensure that workstation caches stay coherent, both so that a write on one workstation is immediately visible to a read on another workstation, and so that complex operations (like creating a file) are atomic even if other workstations are trying to look at the file or directory involved. This last situation is tricky for Frangipani because there's no designated file server that executes all operations on a given file or directory.

In contrast, GFS doesn't have caches at all, since its focus is sequential read and write of giant files that are too big to fit in any cache. It gets high performance for reads of giant files by striping each file over many GFS servers. Because GFS has no caches, GFS does not have a cache coherence protocol. Because the file system logic is in the servers, GFS clients are relatively simple; only the servers need to have locking and worry about crash recovery.

Frangipani appears as a real file system that you can use with any existing workstation program. GFS doesn't present as a file system in that sense; applications have to be written explicitly to use GFS via library calls.

Q: Why do the Petal servers in the Frangipani system have a block interface? Why not have file servers (like AFS), that know about things like directories and files?

One reason is that the authors developed Petal first. Petal already solved many fault tolerance and scaling problems, so using it simplified some aspects of the Frangipani design. And this arrangement moves work from centralized servers to client workstations, which helps Frangipani performance scale well as more workstations are added.

However, the Petal/Frangipani split makes enforcing invariants on file-system-level structures harder, since no one entity is in charge. Frangipani builds its own transaction system (using the lock service and Frangipani's logs) in order to be able to make complex atomic updates to the file system stored in Petal.

Q: Can a workstation running Frangipani break security?

A: Yes. Since the file system logic is in the client workstations, the design places trust in the clients. A user could modify the local Frangipani software and read/write other users' data in Petal. This makes Frangipani unattractive if users are not trusted. It might still make sense in a small organization, or if Frangipani ran on separate dedicated servers (not on workstations) and talked to user workstations with a protocol like NFS.

Q: What is Digital Equipment Corporation (DEC)?

A: It's the company at which the authors worked. DEC sold computers and systems software. Unix was originally developed on DEC hardware (though at Bell Labs, not at Digital).

Q: What does the comment "File creation takes longer..." in Section 9.2 mean?

A: Because of logging, all meta-data changes have to be written to Petal twice: once to the log, and once to the file system. An operation must be written to the log first (thus "write-ahead log"). Then the on-disk file system structures can be written. Only after that can the operation be deleted from the log. That is, a portion of the log can only be freed after all the on-disk file system updates from the operations in that portion have been written to Petal.

So when Frangipani's log fills, Frangipani has to stop processing new operations, send modified blocks from its cache to Petal for the operations in the portion of the log it wants to re-use, and only then free that portion and resume processing new operations.

You might wonder why increasing the log size improved performance. After all, Frangipani has to perform the updates to the file system in Petal no matter how long the log is, and those updates seem to be the limiting performance factor. Here's a guess. It's probably the case that the benchmark involves updating the same directory many times, to add new files to that directory. That directory's blocks will have to be written to Petal every time part of the log is freed (assuming every creation modifies the one directory). So letting the log grow longer reduces the total number of times that directory has to be written to Petal during the benchmark, which decreases benchmark running time. This effect is called "write absorbtion."

Q: Frangipani is over 20 years old now; what's the state-of-the-art in distributed file systems?

A: While some organizations today store user and project files on distributed file systems, their importance has waned with the rise of laptops (which must be self-contained) and commercial cloud services. Still, there are many users of existing file-system protocols such as SMB, NFS, and AFS, as well as more recent distributed file-systems like xtreemfs, Ceph, Lustre, and Dropbox. A huge amount of network storage is sold by companies like NetAPP and EMC, but I don't know to what extent people use the storage at the disk level (via e.g. iSCSI) or as file servers (e.g. talking to a NetAPP server with NFS).

The rise of web sites, big data, and cloud computing have shifted the focus of storage system development. Web sites work well with database-like servers, including key/value stores; file servers are not a good match here. Big data processing systems often use file systems, e.g. GFS for MapReduce, but the focus is on high parallel through for huge files; Frangipani's caching, coherence, and locking are not needed.

Q: The paper says Frangipani only does crash recovery for its own file system meta-data (i-nodes, directories, free bitmaps), but not for users' file content. What does that mean, and why is it OK?

A: If a user on a workstation writes some data to a file on Frangipani, and then the workstation immediately crashes, the recently written data may be lost. That is, if the user logs into another workstation and looks at the file, the recently written data may be missing.

Ordinary Unix file systems (e.g. Linux on a laptop) have the same property: file content written just before a crash may be lost.

Programs that care about crash recovery (e.g. text editors and databases) can ask Unix to be more careful, at some expense in performance. In particular, an application can call fsync(), which tells Unix to force the data to disk immediately so that it will survive a crash. The paper mentions fsync() in Section 2.1.

The rationale here is that the file system carefully defends its own internal invariants (on its own meta-data), since otherwise the file system might not be usable at all after a crash. But the file system leaves maintaining of invariants in file content up to applications, since only the application knows which data must be carefully (and expensively) written to disk right away.

Q: What's the difference between the log stored in the Frangipani workstation and the log stored on Petal?

The Petal paper hardly mentions Petal's log at all. My guess is that Petal logs changes to the mapping from logical block number to physical block number, changes the physical block free list, and changes to the "busy" bits that indicate which block updates the other replica of the block may be missing. That is, Petal logs information about low-level block operations.

Frangipani logs information about file system operations, which often involve updating multiple pieces of file system state in Petal. For example, a Frangipani log entry for deleting a file may say that the delete operation modified some block and i-node free bitmap bits, and that the operation erased a particular directory entry.

Both of the logs are stored on Petal's disks. From Petal's point of view, Frangipani's logs are just data stored in Petal blocks; Petal does not know anything special about Frangipani's logs.

Q: How does Petal take efficient snapshots of the large virtual disk that it represents?

A: Petal maintains a mapping from virtual to physical block numbers. The mapping is actually indexed by a pair: the virtual block number and an epoch number. There is also a notion of the current epoch number. When Petal performs a write to a virtual block, it looks at the epoch number of the current mapping; if the epoch number is less than the current epoch number, Petal creates a new mapping (and allocates a new physical block) with the current epoch. A read for a virtual block uses the mapping with the highest epoch.

Creating a snapshot then merely requires incrementing the current epoch number. Reading from a snapshot requires the epoch number of the snapshot to be specified; then each read uses the mapping for the requested virtual block number that has the highest epoch number <= the snapshot epoch.

Have a look at Section 2.2 of the Petal paper:

http://www.scs.stanford.edu/nyu/02fa/sched/petal.pdf

Q: The paper says that Frangipani doesn't immediately send new log entries to Petal. What happens if a workstation crashes after a system call completes, but before it send the corresponding log entry to Petal?

A: Suppose an application running on a Frangipani workstation creates a file. For a while, the information about the newly created file exists only in the cache (in RAM) of that workstation. If the workstation crashes before it writes the information to Petal, the new file is completely lost. It won't be recovered.

So application that looks like

```
  create file x;
  print "I created file x";
```

might print "I created file x", but (if it then crashes) file x might nevertheless not exist.

This may seem unfortunate, but it's fairly common even for local disk file systems such as Linux. An application that needs to be sure its data is really permanently saved can call fsync().

Q: What does it mean to stripe a file? Is this similar to sharding?

A: It's similar to sharding. More specifically it means to distribute the blocks of a single file over multiple servers (Petal server, for this paper), so that (for example) the file's block 0 goes on server 0, block 1 goes on server 1, block 2 goes on server 0, block 3 goes on server 1, &c. One reason to do this is to get high performance for reads of a single big file, since different parts of the same file can be read from different servers/disks. Another reason is to ensure that load is balanced evenly over the servers.

Q: What is the "false sharing" problem mentioned in the paper?

A: The system reads and writes Petal at the granularity of a 512-byte block. If it stored unrelated items X and Y on the same 512-byte block, and one workstation needed to modify X while another needed to modify Y, they would have to bounce the single copy of the block back and forth between them. Because there's not a way to ask Petal to write less than 512 bytes at a time. It's sharing because both workstations are using the same block; it's false because they don't fundamentally need to.

Q: I donâ€™t understand the section 4 text about how Frangipani enforces the stronger condition of never replaying a completed update.

Here's an example of the problem. Suppose workstation WS1 creates file xxx, then deletes it. After that, a different workstation WS2 creates file xxx. Their logs will look like this:

```
WS1:  create(xxx)  delete(xxx)
WS2:                          create(xxx)
```

At this point, correctness requires that xxx exist (since WS2's create came after WS1's delete).

Now suppose WS1 crashes, and Frangipani recovery replays WS1's log. Recovery will see the delete(xxx) operation in the log. We know that it would be incorrect to actually do the delete, because we know that WS2's create(xxx) came after WS1's delete(xxx). But how does Frangipani's

recovery software conclude that it should ignore the delete(xxx) when replaying WS1's log?

Section 4 is saying that Frangipani's recovery software knows to ignore the delete(xxx) because it "never replays a log record desccribing an update that has already been completed." It achieves this property by keeping a version number in every block of meta-data, and in every log entry; if the log entry's version number is <= the meta-data block's version number, Frangipani recovery knows that the meta-data block has already been updated by the log entry, and thus that the log entry should be ignored.

Q: How much performance improvement could Frangipani get from having a separate lock for each block in each file?

A: It would probably be bad for performance. It would presumably mean that reading a file would require acquiring a lock for each block -- thus much more work for the workstation and the lock server. And there might be no upside in practice, because there are few applications that do read/write sharing at the level of file blocks, and thus could benefit from different workstations being able to modify different blocks of the same file at the same time. It's much more common to see sharing at the level of whole files -- as when your editor writes out the whole file, and then your compiler reads the whole file.

Q: Frangipani and two-phase commit systems both provide distributed transactions. Are they equivalent -- can one view Frangipani as having two-phase commit buried in it? Or are there significant differences?

Frangipani and two-phase commit have similarities. In both cases, in order to perform an operation, all the relevant locks must be acquired, all the relevant data must be gathered, everything must be logged, and only then can locks be released. The big difference is where these things happen. In Frangipani, the locks and data are movable, and are collected on the Frangipani server that's executing the operation. In two-phase commit, the data (with associated locks and logs) are sharded over the servers, so the work has to be split up the same way.

A significant difference is that Frangipani has a story for how to proceed if a server crashes while holding locks -- each Frangipani server's log is in Petal, so another server can recover for it. Classical two-phase commit places a server's log on its local disk, so it's not accessible if the server crashes, so progress is not possible (and locks can't be released) until the server comes back to life.

Another approach to recovery is to replicate each participating two-phase-commit server (including locks, logs, and data) using e.g. Paxos, effectively eliminating crashed servers as a concern. Spanner does this.