

Problem 1 : A bunch of distributed filesystems questions.

- (a) The content of the file `f1` will be since the last write occurred on client 3.
- (b) The content of the file will be since client 4 closed the file last.

Assume a network failure occurred between t_6 and t_7 and it never recovered (i.e., the `close` system call failed on client 4).

- (c) The content of the file `f1` will be since the last write occurred on client 3.
- (d) The content of the file will be either depending whether the message from client 1 will arrive before the message from client 3.

Assume a network failure occurred between t_4 and t_5 and it never recovered (i.e., the `write` failed on client 3 and `close` failed on clients 1, 2, and 4).

Here, we assume that all operations *at* t_4 finish successfully and then the server crashes.

- (e) The content of the file `f1` will be since the last write occurred on client 1.
- (f) The content of the file will be .

Now assume that the server went down between t_4 and t_5 , but everything else was functioning well. It took sufficiently long for the server to come back up so that all the clients finished their applications (since they didn't handle a failed `write` and `close` system calls) happily believing their data was saved.

- (g) Since NFS is stateless and the calls after t_4 didn't get to the server, the content of the file will be .
- (h) The calls didn't get to the server and the applications finished before the server got up again, the content of the file is .
- (i) There are many possible answers. Let's define our metric of goodness how well the file systems scale i.e., how many clients they can support. With NFS's write through policy, the server sees every write that occurs to any file. Therefore, the server can get swamped with lots of traffic and writes propagating all the way to the disk. The AFS server, on the other hand, is able to handle many more clients since files are updated only when the clients close them.
- (j) Unlike AFS, NFS does not provide a global namespace. The mount point on each client is determined by the client's administrator. So you (the user) couldn't necessarily go to `/nfs/home/griffin2` and find your files, whereas you can go to `/afs/ece.cmu.edu/usr/griffin2` from any AFS machine and find your files.
- (k) Idempotency is the property that an action will have the same effect whether it is sent once or multiple times. An idempotent operation for a bank account is "set the balance to \$500"; a non-idempotent operation is "decrement the balance by \$25". For distributed file systems, operations like "set permissions" or "read X bytes from offset Y" are idempotent, whereas "move", "delete", and "read the next X bytes" are non-idempotent.

- (l) An AFS callback is used to invalidate a client's locally-cached copy of some data. There is no NFS callback because NFS servers are stateless; there is no way for them to map which clients hold which blocks. An advantage would be perhaps better cache coherency amongst NFS clients caching the same data (but doing this would mean a fundamental change in NFS semantics).
- (m) There are many wins. Scalability, collaboration, resource sharing, client mobility, centralized administration, to name a few.

Problem 2 : buzzwordsgalore.com.

- (a) The main problem is that the each parity block is stored on the same disks as one of the blocks the parity it contains has been computed over (i.e. one block is always on the same disk and in the same parity group as the parity block). This means that in the case of a disk failure data will be lost – the parity!
- (b) For this problem I wanted you to come up with schemes taking advantage of the properties of the specific RAID scheme in question. Many people came up with general approaches such as using NVRAM or logging, while not incorrect the question was trying to get at something deeper.

For RAID 1: One possible scheme is to order the writes in such a way as to ensure that one mirrored copy is always completely written before the other. One can then check that to see if both copies are the same and if so the write is in a consistent state. If one wants to be able to recover (from for example: a failure that occurs while writing out the second copy) then one could use some type of ECC or checksum written out on completion of the first copy, so the first copy can be detected as complete and mirrored to the second disk.

For RAID 5: In a similar manner one should order writes such that all the data is written out first, which would be compared to the parity to discover whether the data and parity were successfully written out. If one wants to recover from a partial write some type of scheme similar to the one above is necessary.

- (c) To find the $MTTF_{RAID1}$ one can use the formula given in class: $MTTF_{RAID1} = \frac{(MTTF_{Disk})^2}{2 \times MTTF_{Disk}}$, or one can derive it oneself.

Given that replacing a disk takes 3 hours, which is $\frac{1}{2920}$ of a year and that there is a 5% chance per year of failure (the MTTF of a disk is 20 years), this gives a $MTTF_{RAID1} = \frac{20^2}{2 \times \frac{1}{2920}} = 584,000$ years.

- (d) (i) Standard RAID 5 Array (N=8, G=8): $MTTF = \frac{(300,000)^2}{8 \times 7 \times 3} = 61,100$ years
 Two Standard RAID 5 Arrays (N=4, G=4): $MTTF = \frac{(300,000)^2}{4 \times 3 \times 3} \cdot \frac{1}{2} = 143,000$ years
 RAID with Declustered Parity (N=8, G=4): $MTTF = \frac{(300,000)^2}{8 \times 3 \times 3} = 143,000$ years

(ii) Standard RAID 5 Array:

- Single block read (no failures): 1
- Single block write (no failures): 2 disks (each touched twice)
- Single block read (1 failure): 7
- Max failed disks without data loss: 1

Two Standard RAID 5 Arrays:

- Single block read (no failures): 1
- Single block write (no failures): 2 disks (each touched twice)
- Single block read (1 failure): 3
- Max failed disks without data loss: 2

RAID with Declustered Parity:

- Single block read (no failures): 1
- Single block write (no failures): 2 disks (each touched twice)
- Single block read (1 failure): 3
- Max failed disks without data loss: 1

- (iii) The third scheme (Declustered Parity) balances the load most evenly in the case of a single disk failure over multiple RAID. The declustered parity array is constructed with parity groups spanning many overlapping subsets of disks, which helps balance load. For example in the standard multiple array (scheme 2) if disk 2 should fail, each read generates requests to disks 0, 1, and 3 and no accesses to disks 4, 5, 6, or 7. In the declustered parity array, given the same scenario as above, requests are generated to disks 4, 5, and 7 one quarter of the time; to disks 0, 1, and 3 half of the time; and to disk 6 three-quarters of the time.

Standard RAID 5 (scheme 1) has the disadvantage that every disk will be accessed during the read of a failed disk. Even though this seems to evenly distribute the load, more requests are generated thus the overall load is higher.

Problem 3 : AIR RAID.

- (a) (i) Say data is striped across five disks in a RAID 5 array: four disks for data blocks and one disk for parity. If the controller crashes before all four data blocks are written, or before three data blocks and the parity block, then the entire stripe will not be recoverable.
- (ii) RAID controllers can implement RAID scrubbing, during which all of the data blocks are read periodically and their corresponding parity blocks checked. If the parity blocks do not match the data blocks, perhaps due to a media error, then the stripe is reconstructed. In the event of a crash, the entire array could be scrubbed to check for incompletely written stripes. If enough blocks were written then the data could be reconstructed, otherwise the write would have failed.
- (b) A small write will require a data block of a stripe to be written, plus the cost to update the corresponding parity block. This update requires that the parity block be read, modified by the controller, and then re-written to the parity disk. The update is rather simple, the new data can be XORed into the parity block, but the cost of reading, updating, and writing a block will cost at least one full disk rotation, assuming that the disks do not implement writeback caching.
- (c) The benefits of disk arrays are twofold. First, disk arrays allow data to be streamed from multiple disks in parallel, increasing effective bandwidth. In this case, stripe units should be small so that data are striped across as many disks as possible. Second, disk arrays allow for multiple, independent access to unrelated stripes, decreasing access latency. In this case, stripe units should be large so that data are striped across as few disks as possible, to allow for more independent parallel access. Stripe size should be chosen based on the characteristics of the anticipated workload. A workload with more concurrent access would benefit from large stripe units, while a workload with lots of sequential access would benefit from small stripe units.

- (d) It depends on the workload. If the workload includes lots of concurrency, and the stripe unit size is chosen correctly, then queue times should be reduced since independent requests can be serviced in parallel. If the array is optimized for sequential bandwidth, then queue times could increase since concurrent requests will be more likely to access the same drives. If the workload includes lots of sequential requests, and the stripe unit size is selected correctly, then the throughput should increase since requests can be streamed from multiple disks in parallel.
- (e) I would choose RAID 1, straight mirroring. Mirroring has a 100% capacity overhead, since there are two copies of every byte stored, much more than the capacity overhead of RAID 5. Data is highly available in the event of a single disk failure, since no reconstruction is necessary. Transaction rate is high since no parity must be computed on writes, and data can be read from either mirror. However, mean time to failure is lower than that of other RAID schemes.