Distributed Storage on Limping Hardware

Andrew Baptist
Cleversafe, Inc.

Outline

- What is a limping component and why do they matter
- Brief overview of erasure coded storage
- Reading and writing data with limping components

Definition of Limping Components

- A limping component is a component that is working correctly but slowly
- Often due to a hardware failure
- Can also be caused by buggy software or a system "attack"
- Can be transient or permanent (requiring a hardware fix)

Components that limp

- **Disks** - on read errors, desktop drives retry multiple times
- **Controller Cables** - often "smart enough" to negotate lower speeds
- **CPU** - when hot will run in "protection mode" (or if thermometer is broken)
- **Network cable** – bad cables can incorrectly negotiate at lower speeds
- **Network switch**- when overloaded will drop random packets slowing down TCP

Why does it matter?

- Without care, a system will run at the speed of the slowest component
- In large systems limping components are common
- There is not always a clear line between normal and limping operation
  - In some cases, the limping component is only a little slower than normal (e.g. 10%)
  - In the worst case a slow component runs orders of magnitude slower (e.g. 1000%)
- In a storage system there are tradeoffs between reliability, availability and performance

Understanding reliable storage

- Storage devices and disks **will** fail
  - The data that is on them typically can't
- Sites **will** fail
  - Reliable data must be geographically distributed
- For increased reliability, there are 3 options
  - Replication – create a copy (or 2) of the data
  - RAID – create parity disks of the data
  - Erasure coding – RAID++

Why do limping components matter

- When writing to a single disk, the speed of the write depends on the speed of the disk
- With reliable storage, you write to multiple disks
  - Typically when writing to multiple components, the speed of the operation is the speed of the slowest component
- This applies to any reliable storage scheme
  - But the more places you write to, the greater the chance of hitting a limping component

What are the options

- Run the whole system slower when there is a limping component
  - Performance (and possibly availability) implications
- Drop limping components from the system - Fail fast
  - Reliability and availability implications
  - Still have to decide when to mark a component as failed
- Route around the limping component
  - Use it selectively to make tradeoffs between reliability, availability and performance

Dropping limping disks

- When a disk is having problems, it usually will not recover
  - Due to IO failures or very slow responses
- When we detect this condition, we remove the disk from operation
  - User can choose to “retry” the disk or drop it
- Dropping a disk loses only 4 TB, losing a full store can lose ¼ PB

Erasure coding (EC) refresher

- **Source** - user data that needs to be stored
- **Slice** – piece of erasure coded **source**
- **Client** - component that takes user **sources** and distributes them as **slices**
- **Store** - component that stores **slices**
- **Width** - Number of **slices** are generated from a **source** (for writes)
- **Threshold** - Number of **slices** are required to reconstruct a **source** (for reads)

Storing data using erasure coding

- Define a **width** and **threshold** (e.g. 15, 10) for the data
- To write a **source**, the **client** generates **width slices** and writes them to the **store**
- To read a **source**, the **client** requires any **threshold slices** and reconstructs them into a **source**
- The **store** is responsible for efficiently storing **slices**
- The **client** is responsible reconstructing **sources** efficiently from **slices**
- Similar to TCP over the Internet, the logic for reliability, availability and performance is in the clients, not the router.

Reading data algorithm (simple)

- Send a read request for slices to a random **threshold** number of stores
- Once all the slices are received, reconstruct the source
- Problems with "simple" algorithm
  - What if some stores don’t have the data - e.g. disk failure?
  - What if some stores are slower than others?

Reading data algorithm (read all)

- Send read request to **all** stores, process once first threshold stores have responded
- This has two benefits
  - If there is a slow component, it will not affect the read performance (actually up to 5 slow components)
  - For different users, depending on network topology, the read speed will always be optimal
- But one major drawback
  - Reads are unnecessarily sent to extra stores slowing them down and degrading total throughput

Reading data algorithm (ranking)

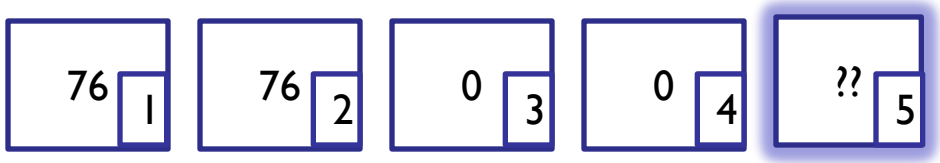
- Keep historical data of past read operations
- Send a read request to the fastest threshold stores
- Occasionally probe a "slow" store to check if it is now faster
- If any request fails (missing data, store down…), send an additional request to other stores
- But… Past performance is not a guarantee of future results

Pessimistic read

- Use the read ranking algorithm from previous slide
- But if any request is too far outside its "standard deviation", choose additional stores
  - In practice, we have found sending an additional request after 3x average response times is a reasonable compromise
- If the slow request returns, use it, but otherwise we can use the one from the new request

Inconsistent revisions

- When reading data, it is possible that not all stores have all data



- **Why** don't stores 3 and 4 have the data?
  - Did we catch it mid-commit?
  - Was the store down when the write occurred?
- Should we **wait** for store 5 or **retry** store 4?
  - Do we retry all stores, or just that one?

Reading data algorithm (final)

- Keep historical data of past read operations
- Send a read request to the fastest threshold stores
- If any request fails, then send an additional request to other stores
- If any response is slow, then send an additional request to other stores
- If global response is inconsistent, then retry operation rather than waiting for a slow store

Comparing reads to writes

- Writing data on limping hardware is harder than reading data
- For writes, if data is not synchronously written to all disks, it is not as reliable
  - This applies to replication, RAID and EC
  - For reads, there is no “long term” impact of choosing which stores to read from
- If data is not written to some disks, it needs to be recomputed later which is expensive (for RAID and EC)

Writing data algorithm (simple)

- Generate slices for all stores
- Send the slices to all stores, wait for a success response
- Once all stores report write success - send a commit to all stores
- Once all stores report commit success - notify the user

Writing with failures

- If is not technically required to write to all stores
- Declaring success after writing to only threshold is risky - what if a disk fails then
- Introduced a concept of a "write threshold"
  - A number between threshold and width that must be written to before declaring success
  - Generally about halfway between width and threshold

Writing data algorithm (threshold)

- Generate slices for all stores
- Send the slices to all stores in parallel, wait for a response
- Once **write threshold** stores report write success - send a commit to those stores
- Once **write threshold** stores report commit success - notify the user

Dealing with the slower stores

- What happens to the slices for stores that we haven't heard a write response from yet?
  - Keep waiting for those responses, and then send the commit when we hear the write response.
- What happens if we can't even send the slice to the store because its network queue is full?
  - Queue the slices in memory send them when we can.
- What happens if we run out of memory to queue these slices?
  - ???

Dealing with slow stores

- When we don’t have memory to continue to queue the slices, we need a backup plan
- Some options
  - Slow down the system to run at the speed of the slowest store (back where we started)
  - Send the slice "somewhere else" - use an out-of-band mechanism to get “home” later
  - Throw the slice away

Ruthless writes

- General idea – if we have filled up our queue due to a slow, start dropping data to that store
- The more memory the client has, the longer we can delay getting to this point
- Three costs to dropping these slices
  - Temporarily reduced reliability
  - Need to rebuild the data later
  - Can make later reads more difficult
- Need to carefully weigh tradeoffs

Writing data algorithm (final)

- Generate slices for all stores
- Send the slices to all stores in parallel
- Once **write threshold** stores report write success - send a commit to those stores
- Once **write threshold** stores report commit success - notify the user
- Continue sending other slices at lower priority
  - If we can't queue them in memory, drop them
- Also deal with all the retry consistency issues from read conflicts

Simulating limping systems

- Various ways to induce artificial slowdowns
  - Configure the NIC speed slower
  - Induce slowdowns on a disks using DM
  - “Under-clock” the CPU (power savings mode)
  - Add additional background processes
- Different slowdowns have different effects
  - Need to test all of them

Future Work

- More adaptive algorithm for ruthless writes
  - When to slow down client writes vs. drop slow slices
- More intelligent algorithm for sampling slow stores
  - Often stores will have a high variance in results
- More intelligent algorithm about when to send additional reads
  - Take into account historical information – (mean, std dev., expected response range, …)
- More intelligent algorithm for determining when to restart an
  - Balancing the cost tradeoffs between restarting vs. waiting

Conclusion

- It is difficult to create a storage system that can efficiently deal with limping hardware
- Writes are harder than reads - need to take into account both immediate cost and future rebuild cost
- Care has to be taken on all "blocking" operations to consider what the options are and never allow a slow store to cause problems

Questions?