Erasure Codes for Systems

COS 518: Advanced Computer Systems
Lecture 19

Wyatt Lloyd

Things Fail, Let's Not Lose Data

- How?

Things Fail, Let's Not Lose Data

- How?
- Replication
  - Store multiple copies of the data
  - Simple and very commonly used!
  - But, requires a lot of extra storage
- Erasure coding
  - Store extra information we can use to recover the data
  - Fault tolerance with less storage overhead
  - **Today's topic!**

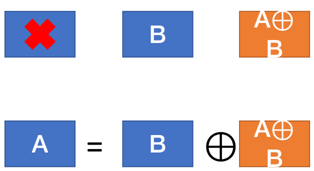
Erasure Codes vs Error Correcting Codes

- Error correcting code (ECC):
  - Protects against **errors** is data, i.e., silent corruptions
  - Bit flips can happen in memory -> use ECC memory
  - Bits can flip in network transmissions -> use ECCs
- Erasure code:
  - Data is **erased**, i.e., we know it's not there
  - Cheaper/easier than ECC
    - Special case of ECC
  - What we'll discuss today and use in practice
    - Protect against errors with checksums

Erasure Codes, a simple example w/ XOR

Erasure Codes, a simple example w/ XOR

Erasure Codes, a simple example w/ XOR



Reed-Solomon Codes (1960)

- N data blocks
- K coding blocks
- M = N+K total blocks

- Recover any block from any N other blocks!

- Tolerates up to K simultaneous failures

- Works for any N and K (within reason)

Reed-Solomon Code Notation

- N data blocks
- K coding blocks
- M = N+K total blocks

- RS(N,K)
  - (10,4): 10 data blocks, 4 coding blocks
    - f4 uses this, FB HDFS for data warehouse does too

- Will also see (M, N) notation sometimes
  - (14,10): 14 total blocks, 10 data blocks, (4 coding blocks)

Reed-Solomon Codes, How They Work

- Galois field arithmetic is the secret sauce

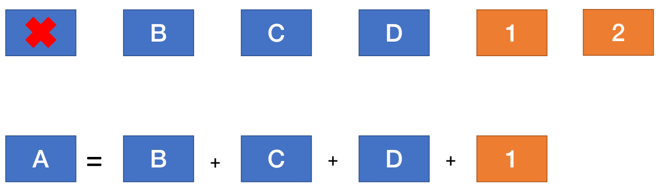
- Details aren't important for us ☺

- See "J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software—Practice & Experience 27(9):995–1012, September 1997."

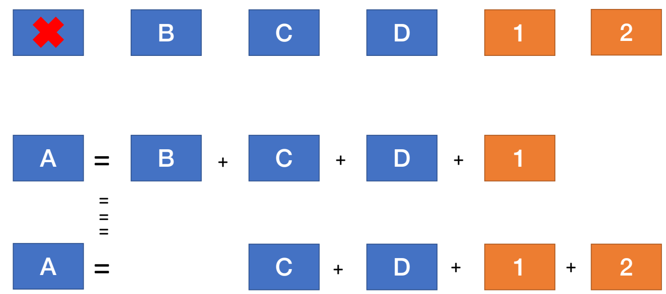
Reed-Solomon (4,2) Example

Reed-Solomon (4,2) Example

Reed-Solomon (4,2) Example



Reed-Solomon (4,2) Example



Erasure Codes Save Storage

- Tolerating 2 failures
  - 3x replication = ___ storage overhead

Erasure Codes Save Storage

- Tolerating 2 failures
  - 3x replication = 3x storage overhead
  - RS(4,2) = ___ storage overhead

Erasure Codes Save Storage

- Tolerating 2 failures
  - 3x replication = 3x storage overhead
  - RS(4,2) = (4+2)/4 = 1.5x storage overhead

Erasure Codes Save Storage

- Tolerating 2 failures
  - 3x replication = 3x storage overhead
  - RS(4,2) = (4+2)/4 = 1.5x storage overhead
- Tolerating 4 failures
  - 5x replication = 5x storage overhead
  - RS(10,4) = ___ storage overhead

Erasure Codes Save Storage

- Tolerating 2 failures
  - 3x replication = 3x storage overhead
  - RS(4,2) = (4+2)/4 = 1.5x storage overhead
- Tolerating 4 failures
  - 5x replication = 5x storage overhead
  - RS(10,4) = (10+4)/10 = 1.4x storage overhead
  - RS(100,4) = ___ storage overhead

Erasure Codes Save Storage

- Tolerating 2 failures
  - 3x replication = 3x storage overhead
  - RS(4,2) = (4+2)/4 = 1.5x storage overhead
- Tolerating 4 failures
  - 5x replication = 5x storage overhead
  - RS(10,4) = (10+4)/10 = 1.4x storage overhead
  - RS(100,4) = (100+4)/100 = 1.04x storage overhead

What's the Catch?

Catch 1: Encoding Overhead

- Replication:
  - Just copy the data
- Erasure coding:
  - Compute codes over N data blocks for each of the K coding blocks

Catch 2: Decoding Overhead

- Replication
  - Just read the data
- Erasure Coding

Catch 2: Decoding Overhead

- Replication
  - Just read the data
- Erasure Coding
  - Normal case is no failures -> just read the data!
  - If there are failures
    - Read N blocks from disks and over the network
    - Compute code over N blocks to reconstruct the failed block

Catch 3: Updating Overhead

- Replication:
  - Update the data in each copy
- Erasure coding
  - Update the data in the data block
  - And all of the coding blocks

Catch 3’: Deleting Overhead

- Replication:
  - Delete the data in each copy
- Erasure coding
  - Delete the data in the data block
  - Update all of the coding blocks

Catch 4: Update Consistency

- Replication:
- Erasure coding

Catch 4: Update Consistency

- Replication:
  - Consensus protocol (Paxos!)
- Erasure coding
  - Need to consistently update all coding blocks with a data block
  - Need to consistently apply updates in a total order across all blocks
  - Need to ensure reads, including decoding, are consistent

Catch 5: Fewer Copies for Reading

- Replication
  - Read from **any** of the copies
- Erasure coding
  - Read from **the** data block
  - Or reconstruct the data on fly if there is a failure

Catch 6: Larger Min System Size

- Replication
  - Need K+1 disjoint places to store data
  - e.g., 3 disks for 3x replication
- Erasure coding
  - Need M=N+K disjoint places to store data
  - e.g., 14 disks for RS(10,4) replication

What’s the Catch?

- Encoding overhead
- Decoding overhead
- Updating overhead
  - Deleting overhead
- Update consistency
- Fewer copies for serving reads
- Larger minimum system size

Many Different Codes

- See “Erasure Codes for Storage Systems, A Brief Primer. James S. Plank. Usenix ;login: Dec 2013” for a good jumping off point
  - Also a good, accessible resource generally

Different codes make different tradeoffs

- Encoding, decoding, and updating overheads
- Storage overheads
  - Best are "Maximum Distance Separable" or "MDS" codes where K extra blocks allows you to tolerate any K failures
- Configuration options
  - Some allow any (N,K), some restrict choices of N and K

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - Updating overhead
    - Deleting overhead
  - Update consistency
  - Fewer copies for serving reads
  - Larger minimum system size

Let's Use Our New Hammer!

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - Updating overhead
    - Deleting overhead
  - Update consistency
  - Fewer copies for serving reads
  - Larger minimum system size

Immutable data

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - ~~Updating overhead~~
    - Deleting overhead
  - ~~Update consistency~~
  - Fewer copies for serving reads
  - Larger minimum system size

Immutable data

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - ~~Updating overhead~~
    - Deleting overhead
  - ~~Update consistency~~
  - Fewer copies for serving reads
  - Larger minimum system size

Immutable data

Storing lots of data
(when storage overhead
actually matters this is true)

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - ~~Updating overhead~~
    - Deleting overhead
  - ~~Update consistency~~
  - Fewer copies for serving reads
  - ~~Larger minimum system size~~

Immutable data

Storing lots of data
(when storage overhead
actually matters this is true)

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - ~~Updating overhead~~
    - Deleting overhead
  - ~~Update consistency~~
  - Fewer copies for serving reads
  - ~~Larger minimum system size~~

Data is stored for a long
time after being written

Immutable data

Storing lots of data
(when storage overhead
actually matters this is true)

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - ~~Updating overhead~~
    - Deleting overhead
  - ~~Update consistency~~
  - Fewer copies for serving reads
  - ~~Larger minimum system size~~

Low read rate

Data is stored for a long time after being written

Immutable data

Storing lots of data (when storage overhead actually matters this is true)

Erasure Coding Big Picture

- Huge Positive
  - Fault tolerance with less storage overhead!
- Many drawbacks
  - Encoding overhead
  - Decoding overhead
  - ~~Updating overhead~~
    - Deleting overhead
  - ~~Update consistency~~
  - ~~Fewer copies for serving reads~~
  - ~~Larger minimum system size~~

Low read rate

Data is stored for a long time after being written

Immutable data

Storing lots of data (when storage overhead actually matters this is true)

f4: Facebook's Warm BLOB Storage System [OSDI '14]

Subramanian Muralidhar*, Wyatt Lloyd*ᵠ, Sabyasachi Roy*, Cory Hill*, Ernest Lin*, Weiwen Liu*, Satadru Pan*, Shiva Shankar*, Viswanath Sivakumar*, Linpeng Tang*⁺, Sanjeev Kumar*

*Facebook Inc., ᵠUniversity of Southern California, ⁺Princeton University

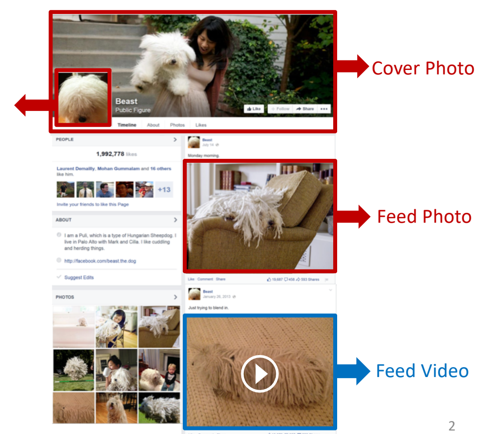
1

BLOBs@FB

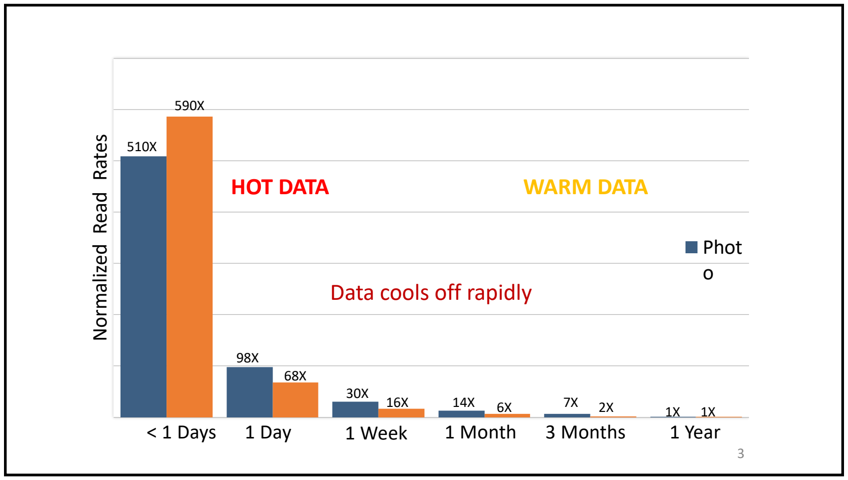
Immutable & Unstructured

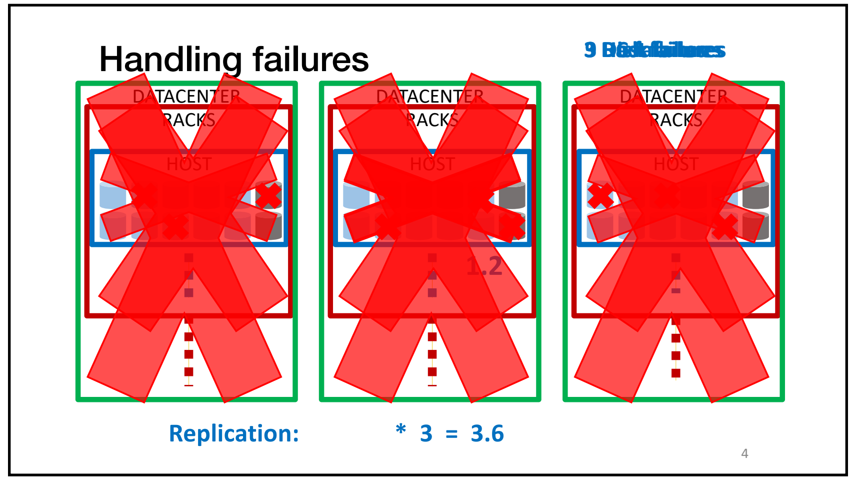
Diverse

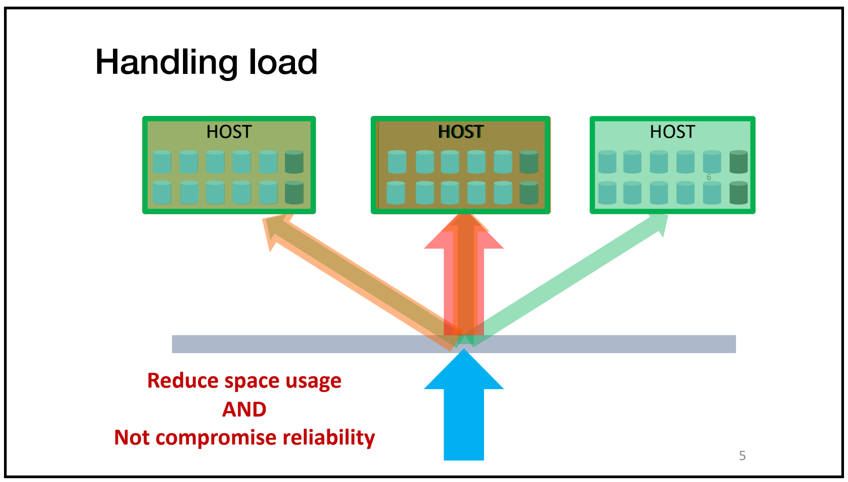
A LOT of them!!

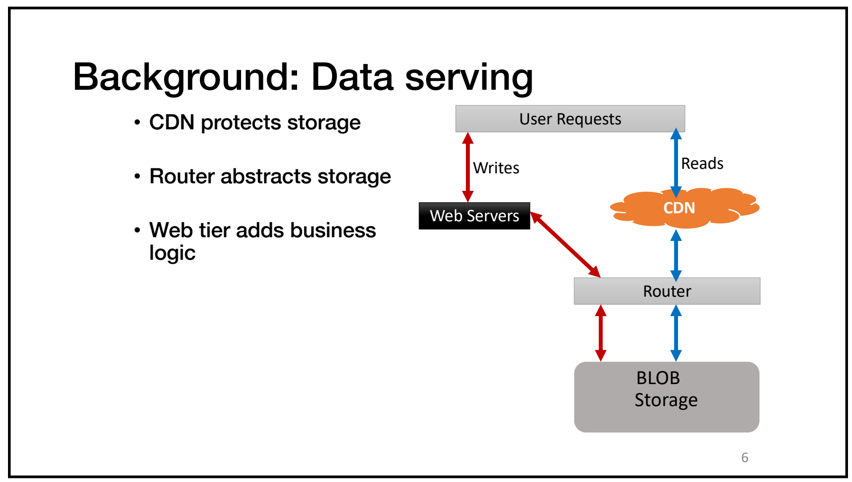


2


Normalized Read Rates
590X
510X
HOT DATA
WARM DATA
Photo
Data cools off rapidly
98X
68X
30X
16X
14X
6X
7X
2X
1X
1X
< 1 Days
1 Day
1 Week
1 Month
3 Months
1 Year
3

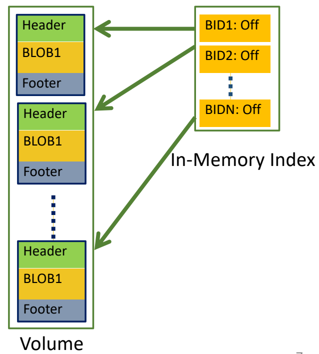

Handling failures
DATACENTER
RACKS
HOST
DATACENTER
RACKS
HOST
DATACENTER
RACKS
HOST
1.2
Replication:
* 3 = 3.6
4


Handling load
HOST
HOST
HOST
Reduce space usage
AND
Not compromise reliability
5


Background: Data serving
• CDN protects storage
• Router abstracts storage
• Web tier adds business logic
User Requests
Writes
Reads
Web Servers
CDN
Router
BLOB
Storage
6

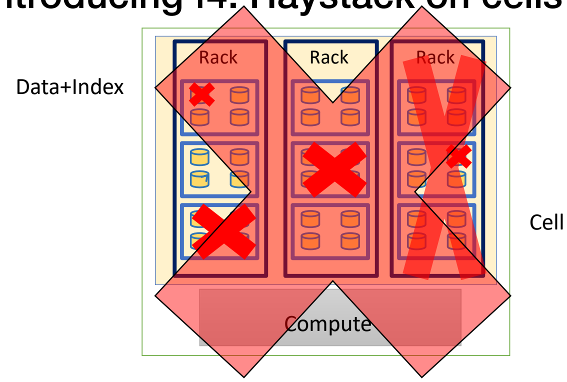
Background: Haystack [OSDI2010]

- Volume is a series of BLOBs
- In-memory index



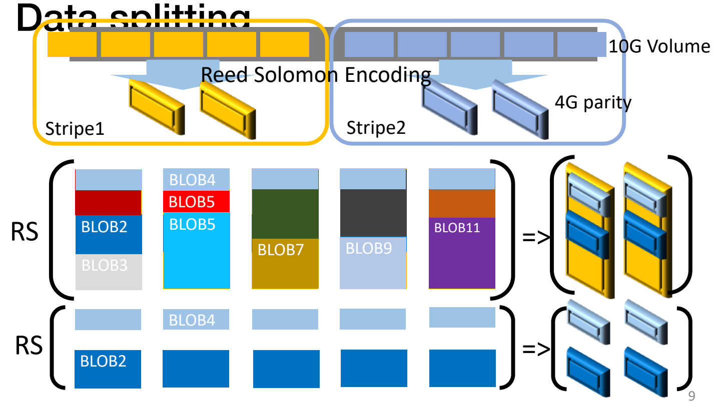
7

Introducing f4: Haystack on cells



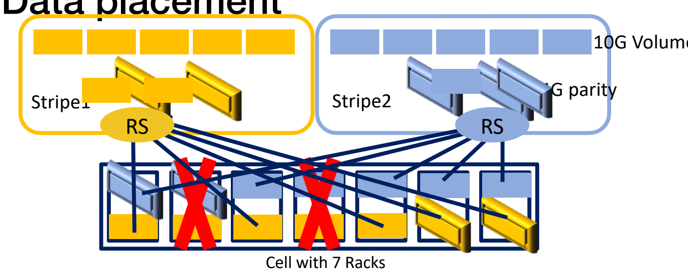
8

Data splitting



9

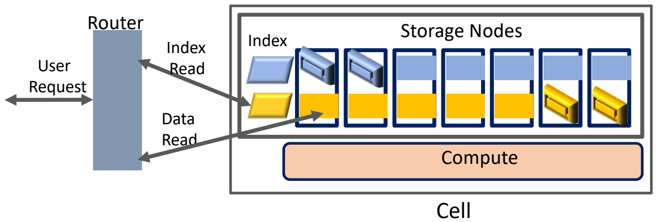
Data placement



- Reed Solomon (10, 4) is used in practice (1.4X)
- Tolerates 4 racks (→ 4 disk/host) failures

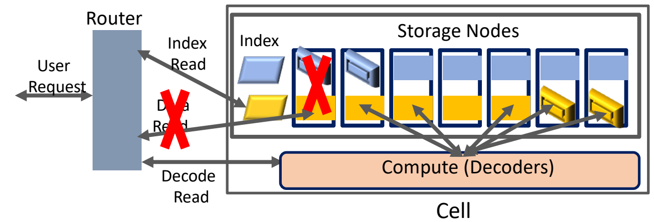
10

Reads



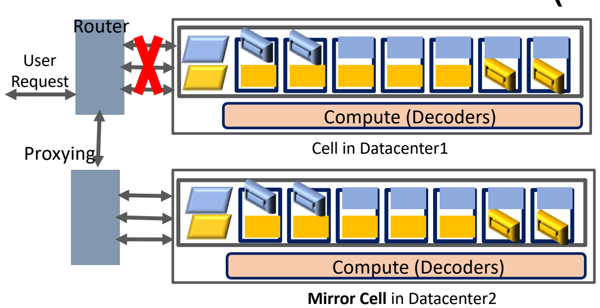
- 2-phase: Index read returns the exact physical location of the BLOB

Reads under cell-local failures



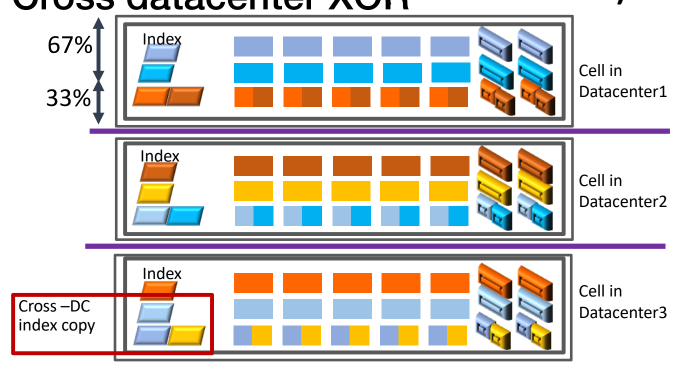
- Cell-Local failures (disks/hosts/racks) handled locally

Reads under datacenter failures (2.8X)

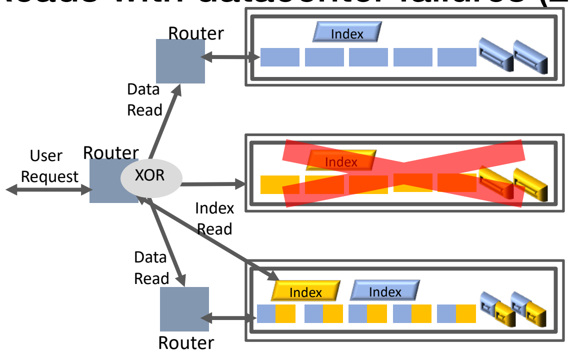


2 * 1.4X = 2.8X

Cross datacenter XOR (1.5 * 1.4 = 2.1X)



Reads with datacenter failures (2.1X)



15

Haystack v/s f4 2.8 v/s f4 2.1

	Haystack with 3 copies	f4 2.8	f4 2.1
Replication	3.6X	*2.8X*	*2.1X*
Irrecoverable Disk Failures	9	*10*	*10*
Irrecoverable Host Failures	3	*10*	*10*
Irrecoverable Rack failures	3	*10*	*10*
Irrecoverable Datacenter failures	*3*	2	2
Load split	*3X*	2X	1X

16

Evaluation

- What and how much data is “warm”?
- Can f4 satisfy throughput and latency requirements?
- How much space does f4 save
- f4 failure resilience

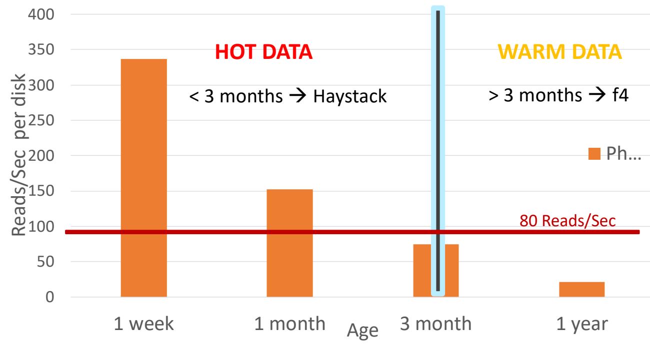
17

Methodology

- CDN data: 1 day, 0.5% sampling
- BLOB store data: 2 week, 0.1%
- Random distribution of BLOBs assumed
- The worst case rates reported

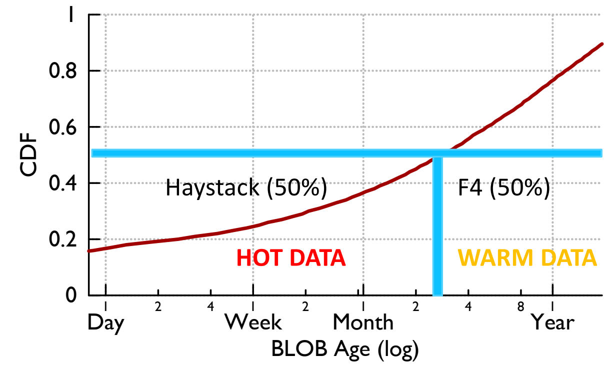
18

Hot and warm divide



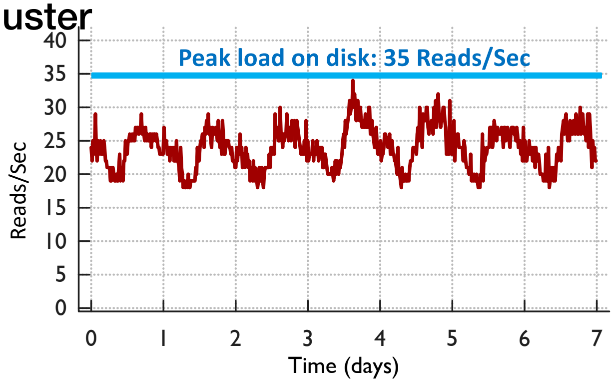
19

It is warm, not cold



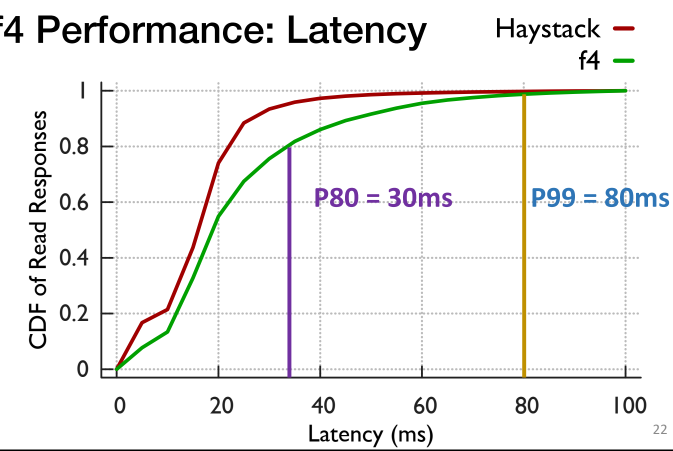
20

f4 Performance: Most loaded disk in cluster



21

f4 Performance: Latency



22

Concluding Remarks

- Facebook's BLOB storage is big and growing
- BLOBs cool down with age
  - ~100X drop in read requests in 60 days
- Haystack's 3.6X replication over provisioning for old, warm data.
- f4 encodes data to lower replication to 2.1X

23

The Akamai Networwrk: A Platform for High-Performance internet Applications

COS 518: *Advanced Computer Systems*

Fei Gao

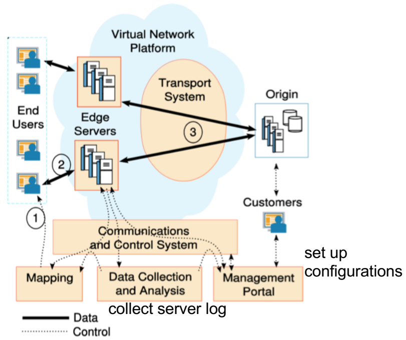
4/17/2018

Problem: Internet Delivery Challenges

- Peering point congestion (middle mile)
- Inefficient routing/communication protocols
- Unreliable networks
- Scalability
- Application limitations and slow rate of change adoption

68

Delivery Network Overview



69

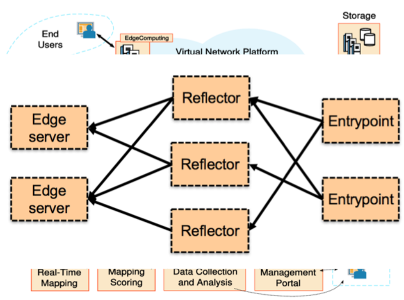
Transport System

- Content and Streaming Media Delivery
- Application Delivery

70

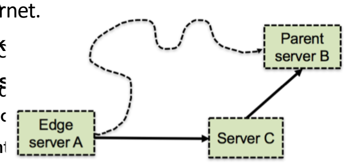
Content Delivery

- Depl
- Thou
- Tiere
- Ovei



Application Delivery

- Speed up long-haul Internet communications by using the Akamai platform as a high-performance overlay network.
- Push application logic from the origin server out to the edge of the Internet.
  - Path
    - Pack
  - Edge
  - Trans
  - Applic
    - Po
    - Int ... ased on the ne
  - databases
- Application optimization



Edge Server Platform

- Origin server location
- Cache control/indexing
- Access control
- Response to origin failure
- EdgeComputing

Mapping System

- Scoring
  - Based on tremendous amounts of historic and real-time data
- Real-time mapping
  - Direct the end user to the best edge server
  - Map to cluster: based on score
  - Map to server: based on the cached content

Result:

- New York Post: 20X performance improvement
- U.S. government: Protection against DDos
- MySpace: 6X speedup, 98% offload
- Haiti Benefit Concert: broadcast it online, 5.8M streams served in a weekend, $61 million raised.

Impact

- Google scholar citation: 605
- Market share: 45% in 2017
- 15%-30% of global network traffic comes from Akamai

• Thank you!

Experiences with CoralCDN: A Five-Year Operational View

COS 518: *Advanced Computer Systems*

Felix Madutsa

April 18 2018

CoralCDN

"You make it fun; we'll make it run"

The Coral Content Distribution Network

- A free, open, P2P and self-organizing web CDN designed by Michael J. Freedman
- No prior registration, authorization, or special configuration needed
- Publishing by appending a suffix to a URL's hostname, *e.g.*, http:/ /example.com.*nyud.net*/
- Designed to automatically and scalably handle sudden spikes in traffic (flash crowds) for new content in services that suffer from overloads
- Deployed on the PlanetLab research network (260 servers) between 2004 - 2015

CoralCDN Usage

"You make it fun; we'll make it run"
CORAL
The Coral Content Distribution Network

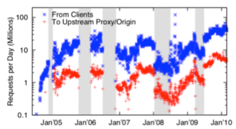


Figure 3: Total HTTP requests per day during CoralCDN's deployment. Grayed regions correspond to missing or incomplete data.

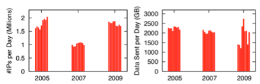
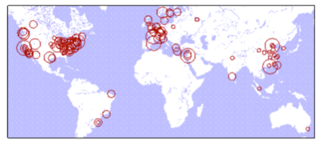


Figure 4: CoralCDN usage: number of unique clients (left) and upload volume (right) for each day during August 9–18.



Year	Unique domains	Unique URLs	% URLs with 1 req	Reqs to most popular URL
2005	7881	577K	54%	697K
2007	21555	588K	59%	410K
2009	20680	1787K	77%	1578K

Figure 5: CoralCDN traffic statistics for an arbitrary day (Aug 9).

Problem Statement/Motivation

- Present data collected over the system's production deployment and its implications
- Discuss deployment challenges encountered and describe preferred solutions
- Insights for building a secure, open, and scalable content distribution network

Web Security Implications of Open API

- CoralCDN as an elastic resource
  - Simple and friendly API allowed wide + dynamic adoption and misuse
- Open API + naming techniques caused security problems from lack of explicitness in specifying protection domains

Security Mechanisms

1. Limiting functionality
   - Brute-force attacks on websites are slow
   - Cannot perform anonymous attacks
2. Reducing excessive resource use
   - Fair sharing mechanisms to balance bandwidth consumption
   - Monitoring of both client-side and server side usage
3. Blacklisting domains and offloading security
   - Global blacklist for phishing attempts, copyright violations, access control violations

Lessons for CDNs

- CoralCDN designed to interact with overloaded or poor-behaving servers

- Unlike commercial CDNs, Coral cannot grow the size of the network best on expected use

Lessons for CDNs

1. Designing for faulty origins
   - Cache negative service results
   - Serve stale content if origin faulty (<= 24hrs)
   - Prevent truncations through whole-file overwrites
   - Decoupling service dependences

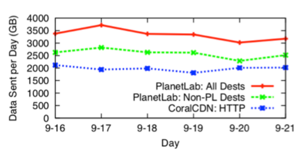
2. Manage oversubscribed bandwidth
   - Respond with "Forbidden" when domain oversubscribed

3. Managing performance jitters
   - Need to guarantee stability

Lessons for Platforms

- Application developments could benefit from greater visibility and control of lower layers

1. Exposing information and expressing preferences across layers
   - Export greater information to higher levels
   - Applications push policies to lower levels



Lessons for Platforms

- Application developments could benefit from greater visibility and control of lower layers

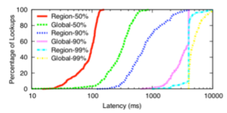
1. Exposing information and expressing preferences across layers
   - Export greater information to higher levels
   - Applications push policies to lower levels

2. Support for fault-tolerance
   - Dynamically update root nameserver to reflect change
   - Announcing IP anycast address via BGP

Insights for Building Open CDNs

- Coral was overdesigned for its workload

1. Naming
   - Levels to support layers of indirection
2. Content Integrity
   - End-to-end signature for content integrity through HTTP
3. Fine-Grain Origin Control
   - Change origin policy



89

Impact?

[PDF] Experiences with CoralCDN: A Five-Year Operational View.
MJ Freedman - NSDI, 2010 - static.usenix.org
Abstract CoralCDN is a self-organizing web content distribution network (CDN). Publishing through CoralCDN is as simple as making a small change to a URL's hostname; a decentralized DNS layer transparently directs browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin webserver. CoralCDN has been publicly available on PlanetLab since March 2004, accounting for the majority of its bandwidth and serving requests for several million users (client IPs) per day. This paper ...
☆ 99 Cited by 101 Related articles All 28 versions

90