


DePaul University

SE480 Week 9 – Case Studies

Steven Engelhardt

DePaul University

Autumn 2020

Table of Contents

Summary of Last Week

- Discussed *Reflection* architectural pattern
- Architecture in the lifecycle

Table of Contents

Disclosure – I Am Not A Twitter Expert

Material generated from:

- http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html
- http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html
- http://highscalability.com/blog/2016/4/20/how-twitter-handles-3000-images-per-second.html
- https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

- https://www.slideshare.net/caniszczyk/twitter-opensourcestacklinuxcon2013
- https://code.facebook.com/posts/1566627733629653/mobile-scale-london-recap/
- http://highscalability.com/blog/2011/12/19/how-twitter-stores-250-million-tweets-a-day-using-mysql.html
- https://www.infoq.com/news/2017/02/twitter-react-mobile-stack
- https://www.infoq.com/presentations/Real-Time-Delivery-Twitter
- https://blog.twitter.com/engineering
- https://www.youtube.com/watch?v=5cKTP36HVgI
- Wikipedia

It's 2006

- It's March 21, 2006. Four friends have an idea for a new form of communication – using a SMS service to comminute with a small group – which will eventually become Twitter.
- The goal is to get something up-and-running to show publicly by July 15, 2006.
- What architectural qualities should be emphasized? Deemphasized?
- What decisions need to be made early, and quickly?
- What might an early architecture look like?

Architectural Qualities

Emphasize:

- Speed-to-market
- Modifiability
- Usability
- Performance?
- Scalability?

Deemphasize:

- Interoperability
- Availability
- Security
- Testability
- Portability
- Performance?
- Scalability?

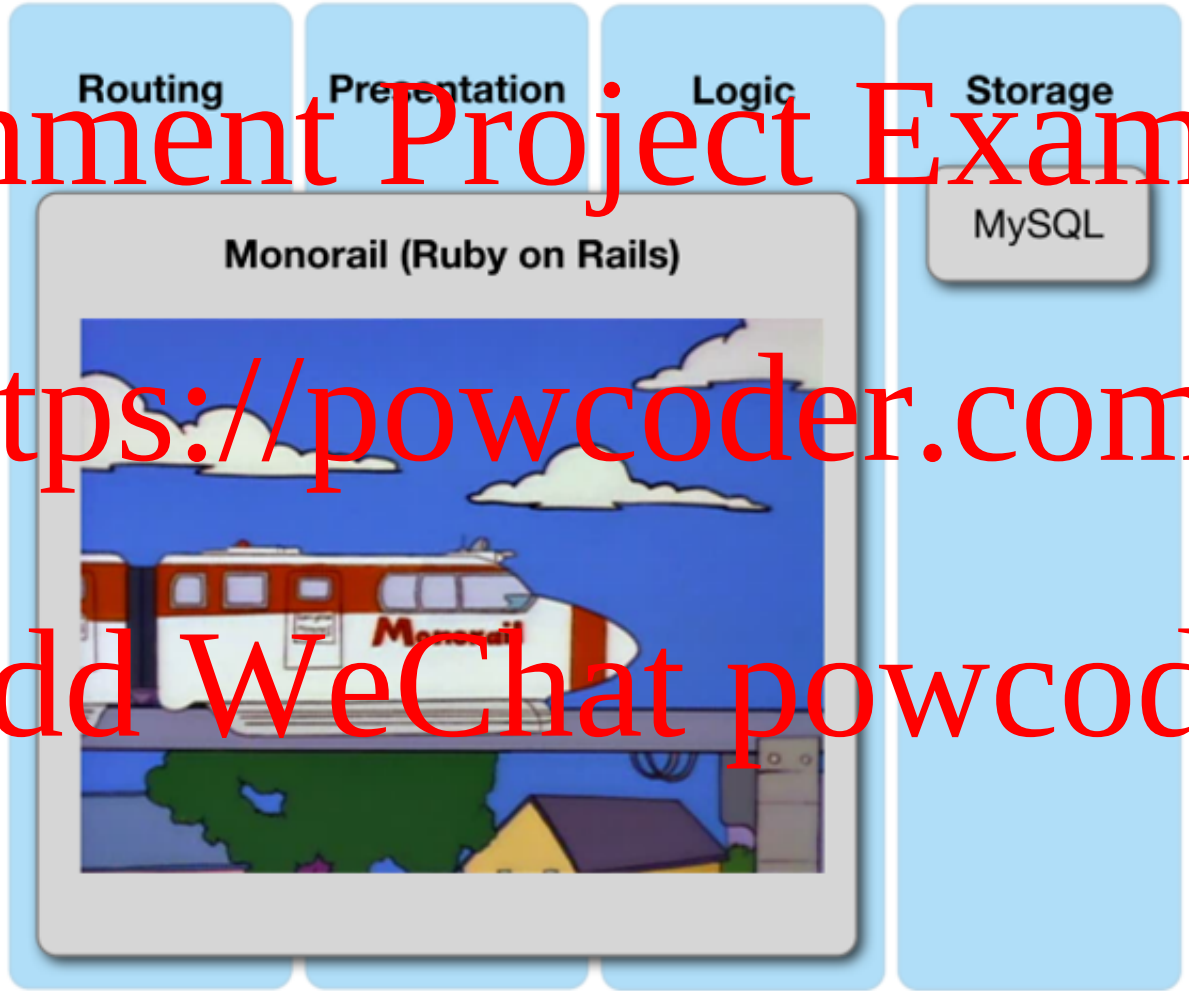
Early Decisions

- What language & platform to use to implement the website? Open source or commercial?
- What database system to use to store and query the tweets?
- Should we support push & pull models?
- Should the system be monolithic? Microservices? Service-oriented? Etc.
- How will users authenticate?
- Where will we host & run the system?
- (many more)

Twitter in One SQL Statement


```
SELECT TOP 10 T.text
FROM tweets T
INNER JOIN user_follows UF
    ON UF.follow_userid = T.userid
WHERE UF.userid = @userid
ORDER BY T.date DESC
```

Twitter Circa 2006



Where Are We?

- We launched an app in 4 months!
- We're easily able to handle our load of 20,000 tweets/day
- Using Ruby on Rails allows us to launch new features quickly & easily
- Performance is not too bad, especially once we've used proper database indexing
- The app is really starting to take off!

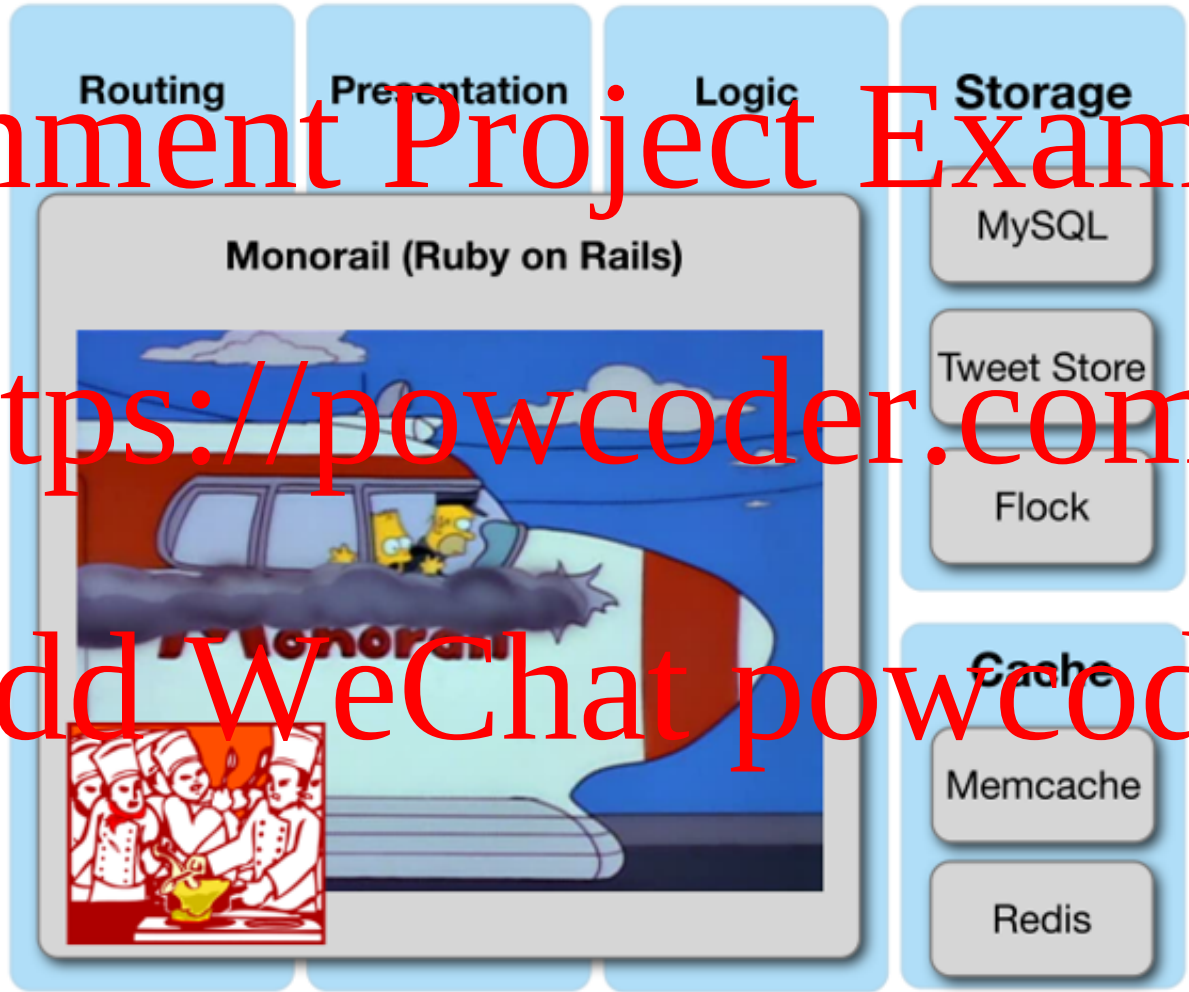
2008 – Database Bottleneck

- In 2007, Twitter handled 400,000 tweets per quarter.
- In 2008, it became 100,000,000 tweets per quarter.
- We can scale out our Ruby on Rails servers, but our single-instance database is starting to melt down. What do we do?

Some Options

- Separate into multiple databases by data domain
  - How to separate?
- Database read replicas
  - Helps with reads, but what about writes?
- Database sharding
  - What shard key?
- Caching
  - What do we cache?
- A custom data storage system?

Twitter Circa 2008



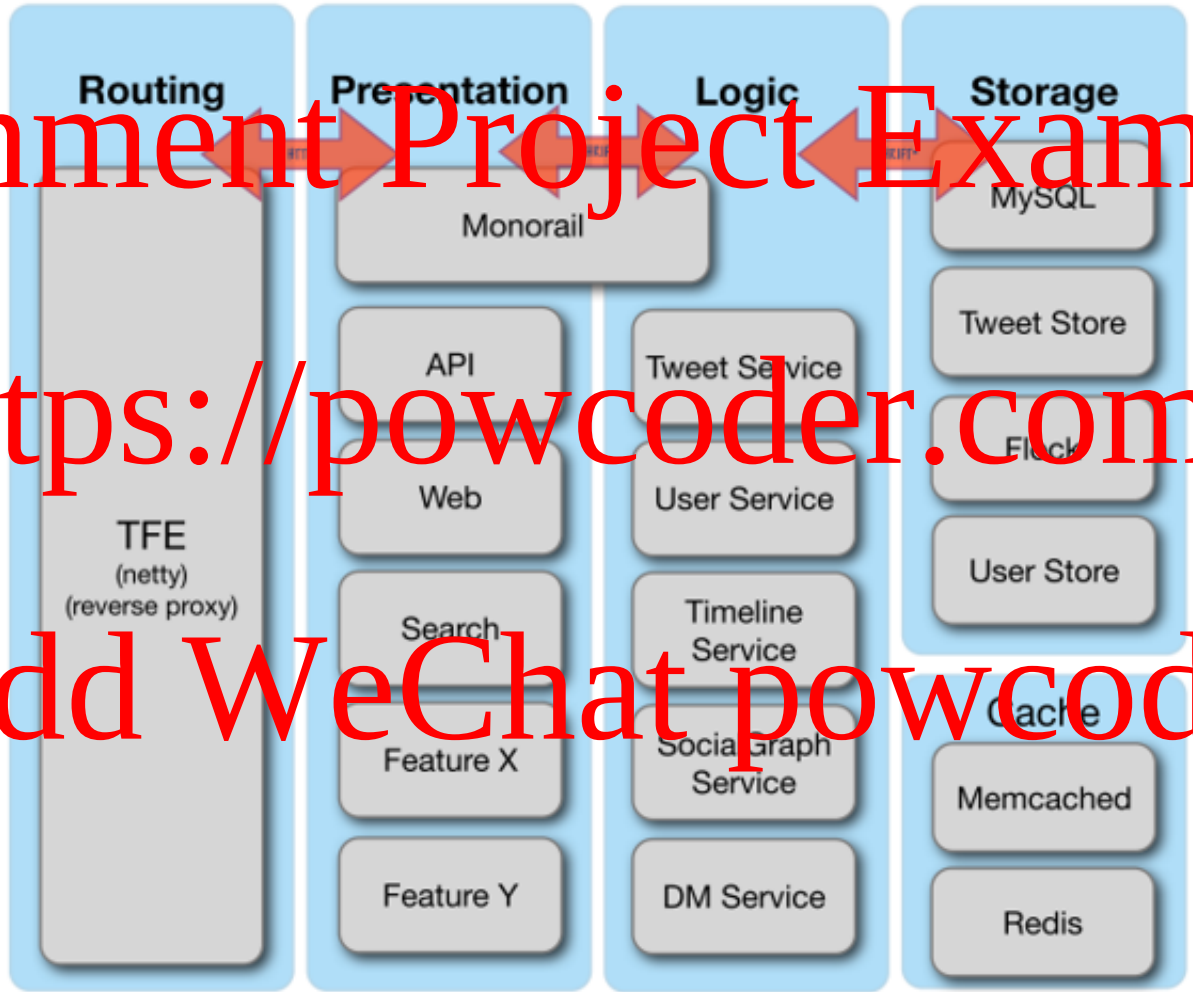
2010 – App Bottlenecks

- The growth won't stop! We're now at 4.5 billion tweets per quarter!
- Our monolithic Rails code base is fragile and hard to maintain
- We're trading off readability & flexibility for performance
- Our Rails machines are having the CPU & RAM maxed but not the network, and it's getting really costly to run so many Rails machines
- What do we do?

Embrace JVM, Decompose the Monolith

- Experimentation with the JVM shows it to be mature, world-class, and *fast* with a huge mature ecosystem of libraries
- Time to decompose the monolith into services based on the core nouns
  - *Tweet* service
  - *User* service
  - *Timeline* service
  - *DM* service
  - *Social Graph* service
- Don't forget about retooling and retraining!

Twitter Circa 2010



2012 – Rethinking Tweet Storage & Querying

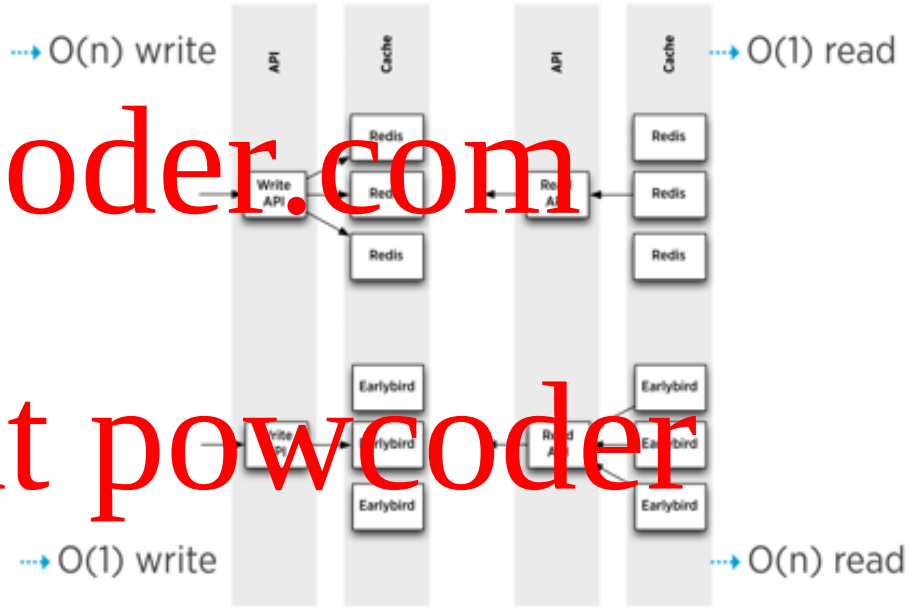
- We're now over 22.5 billion tweets/quarter, with 150M active users worldwide, 300K timeline queries per second, and peaks of over 25,000 tweets/second
- To meet these performance requirements, we need to dramatically rethink how we store tweets.
- What can we do?

Tradeoff Between Read & Write Speed

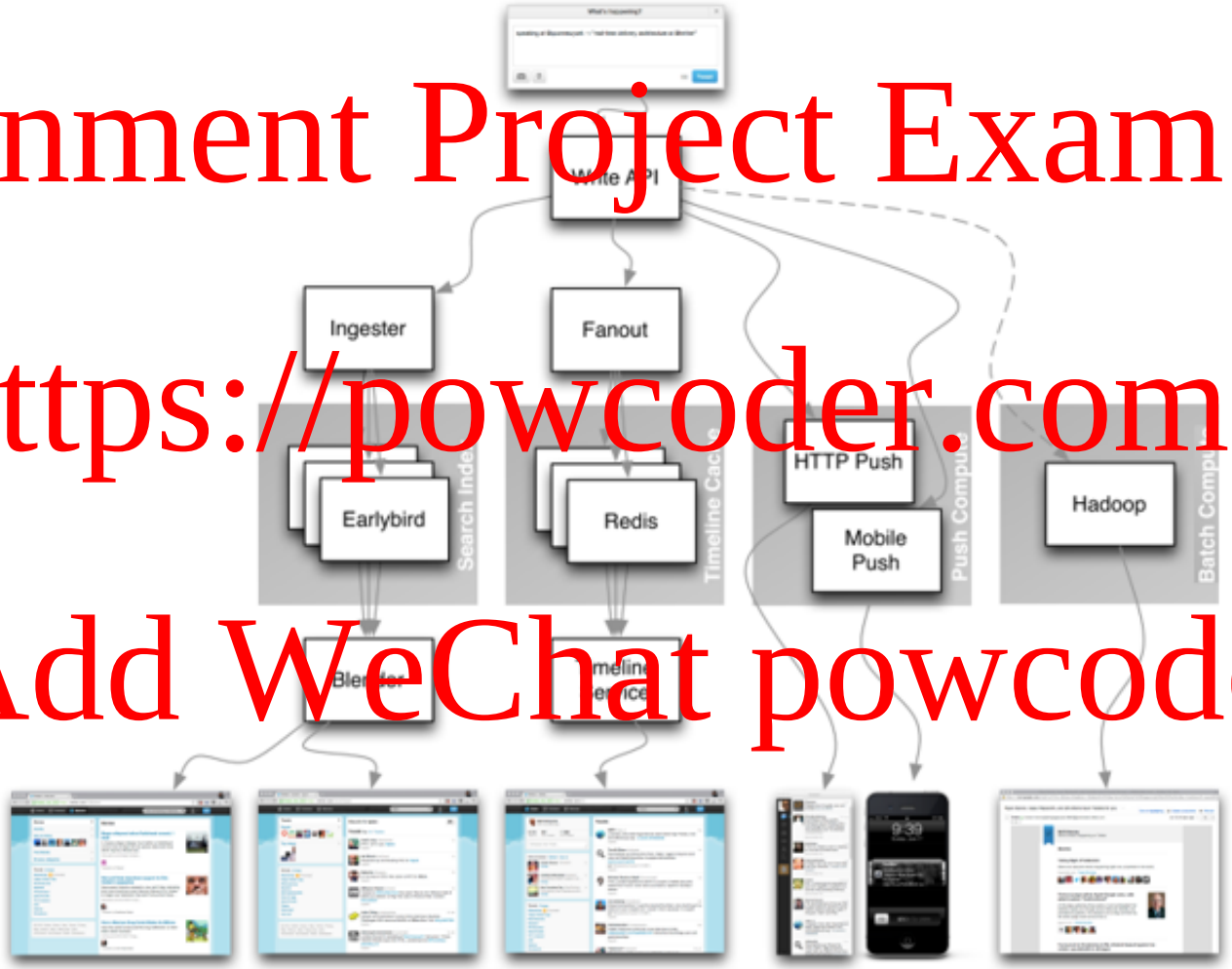
- Architects frequently find that they must tradeoff between read & write speed. In other words, to make reads faster, they must make writes slower, or vice versa.
- The biggest problem with Twitter read/write performance is fanout.
- What if we performed the social network fanout at write time?
  - If we did, how would we handle search?

Fanout at Write Time for Timeline

Scatter-Gather at Read Time for Search



Also Support Push and Analytics



The Challenge

- Fanout can be really slow, especially for high follower counts!
- This can occasionally result in tweets being delivered out-of-order (a user seeing a reply or retweet before seeing the original tweet)
- Maybe the right answer is a combination of fanout and scatter-gather...
  - Will this ever end?

- We need to introduce the ability for Twitter to handle media (images, videos, etc.) at Twitter scale
- What are the main concerns here?
- What might the architecture look like?

Handling Media v1 – Concerns

- Scalability
- Estimated 3,000 images/second, 200GB of images/second
- Delivery to multiple devices, at multiple resolutions (a.k.a. variants)

Handling Media v1 – Architecture

Figure: Write path

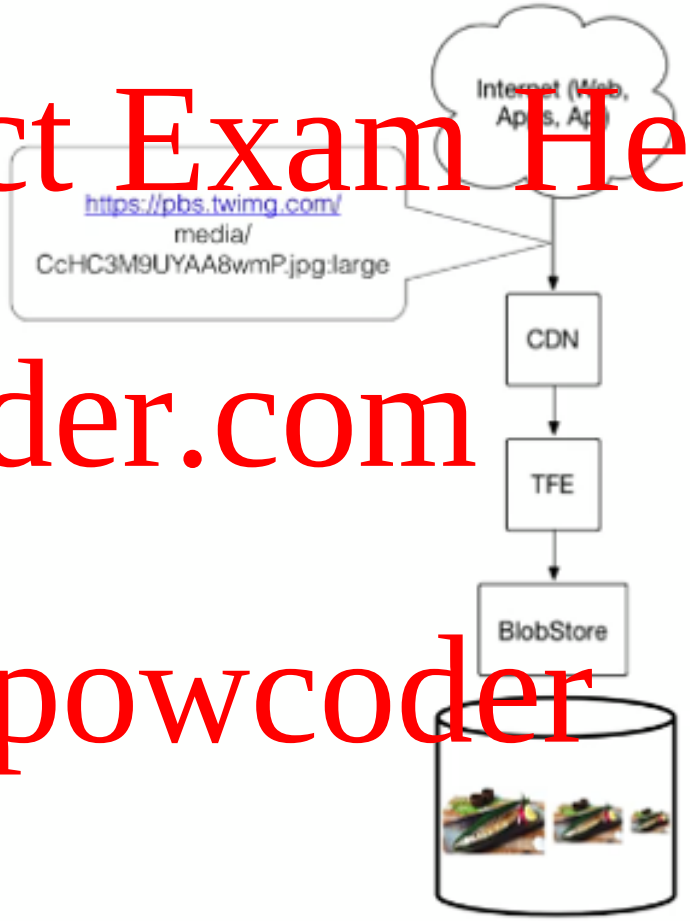


Figure: Read path

Handling Media v1

Advantages

- Fast to read
- Simple

Disadvantages

- Uploads would frequently fail
- Inflexible
  - Variants fixed at time of upload, short of regenerating all possible variants (no variants on demand)
- Bloated image storage
  - No garbage collection of old or unrequested images

- Separate media upload from tweet upload
- Allow multi-part uploads of media file to support restarting in the middle
- Support new image types (e.g. progressive JPEG)
- Perform variant generation at read-time rather than write-time

Handling Media v2 – Architecture

Figure: Write path

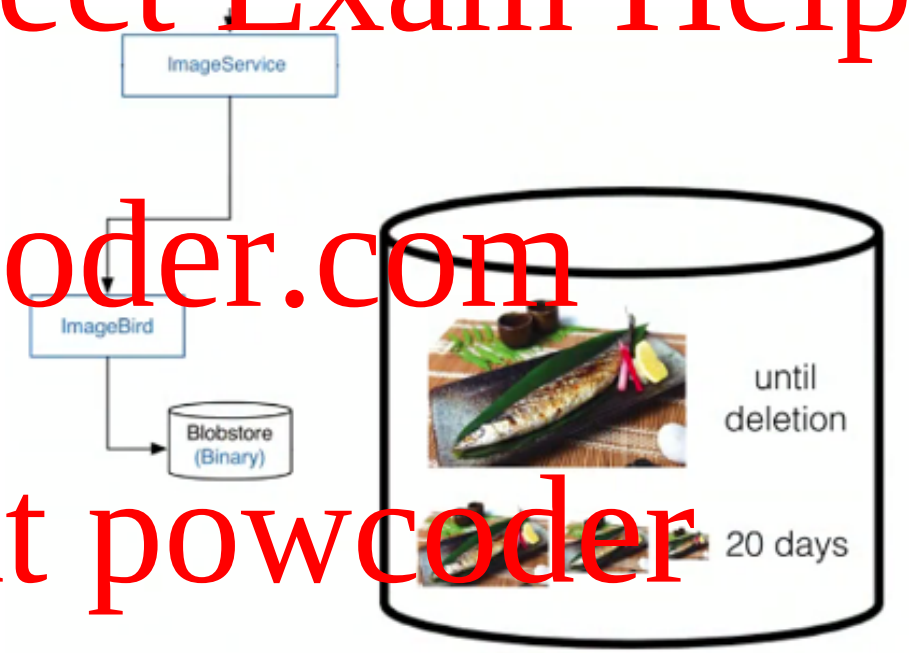


Figure: Read path

Other Interesting Twitter Topics

- Supporting push v. pull models
- Building an API that can be used across all apps
- Real-time and batch analytics
- Monitoring & telemetry
- The evolution of their web stack, and client-side vs server-side rendering

Key Lessons

- Architecture is all about tradeoffs
- Architectural qualities in focus changes ovr the lifecycle of an application
  - This means that the 'right' architecture for a system often changes over time!
- Incremental changes always win
- If your application experiences exponential growth, expect significant rearchitecture!

Table of Contents

What is Git?

- Git is the most popular source code version management tool
- It is the foundation behind the massive code sharing site GitHub
- It was created by Linus Torvalds in 2005 for development of the Linux kernel, but has been maintained since 2005 by Junio Hamano

Initial Goals of Git

- Support distributed workflows similar to those enabled by BitKeeper
- Offer safeguards against content corruption
- Offer high performance
- Philosophy: Be the "anti-CVS" – whenever in doubt of a decision, choose the option that was *not* chosen by CVS
  - More on CVS on the next slide

What is CVS?

- The most popular version control system for about a decade until it was replaced in 2000 by Subversion ("CVS but better") [2]
- One central backend (repository) which holds all projects and all source code history. Nearly all operations require communication with the central backend.
- Each file in source control is versioned separately. There is no notion of a "commit" object.
- The repository uses file-based *delta compression* for efficient storage of different versions of the same file
- Each developer only has a checked-out snapshot of source code, never the full development history

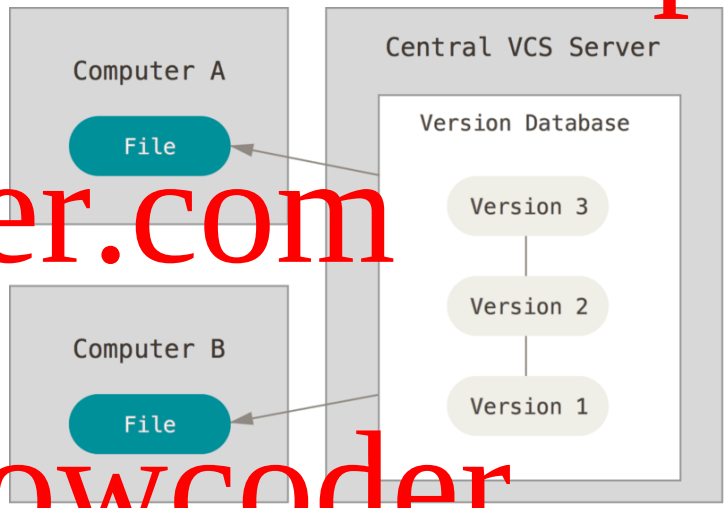


Figure: Centralized version control [1]

Design Characteristics of Git [ ]

- Strong support for non-linear development
  - Rapid and lightweight branching and merging, with the assumption that a change will be merged more often than it is written
- Distributed development
  - Each developer has a local copy of the full development history
- Efficient handling of large projects
- Cryptographic authentication of history
- Pluggable merge strategies
- Garbage accumulates until collected
- Periodic explicit object packing

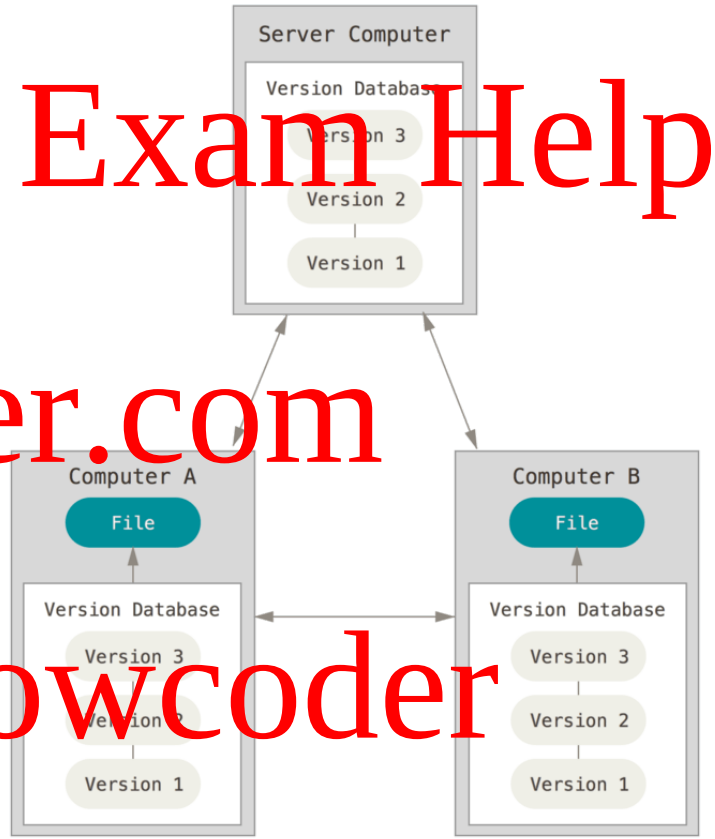


Figure: Distributed version control [1]

Git Stores Snapshots, Not Differences

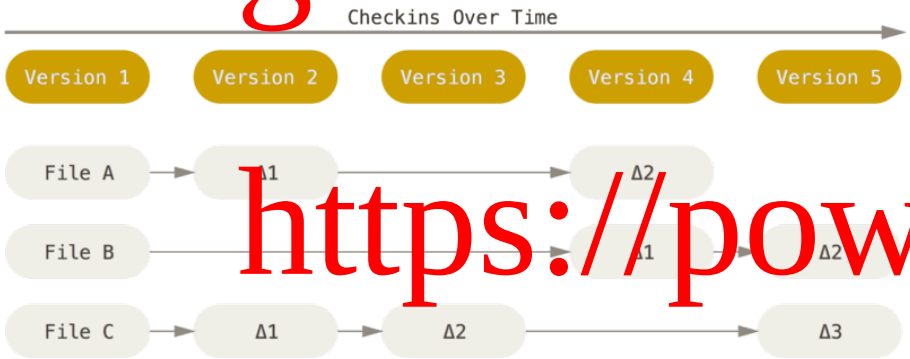


Figure: Storing data as changes to a base version of each file [1]

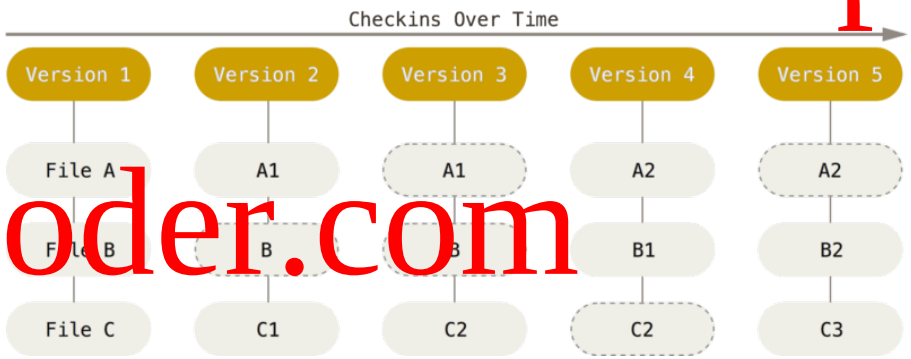


Figure: Storing data as snapshots of the project over time [1]

Git Uses SHA-1 Hashes for Integrity

- Everything in Git is checksummed before it is stored and is then referred to by that checksum
- This means it's impossible to change the contents of any file or directory without Git knowing about it
- The mechanism that Git uses for this checksumming is called a SHA-1 hash
- A SHA-1 hash looks something like this:
  24b9da6552252987aa493b52f8696cd6d3b00373

Everything is SHA-1 Hashed

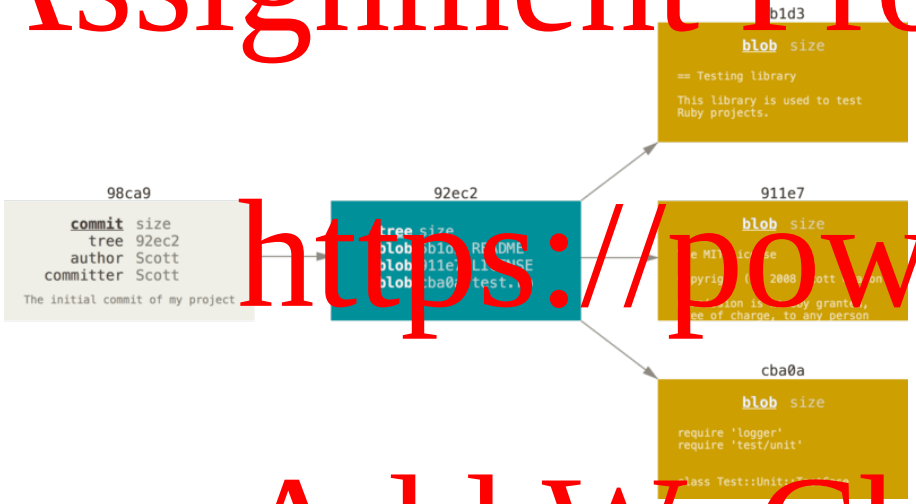


Figure: A commit and its tree [1]

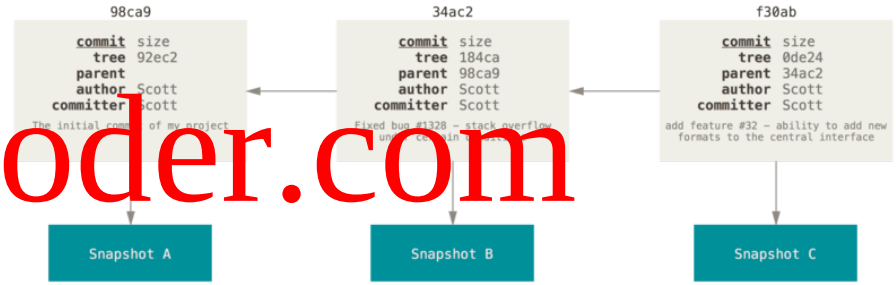


Figure: Commits and their parents [1]

Branches are Pointers

- A branch in Git is simply a lightweight movable pointer to a commit
- `HEAD` is a special pointer which is used to mean "the branch you're currently on"

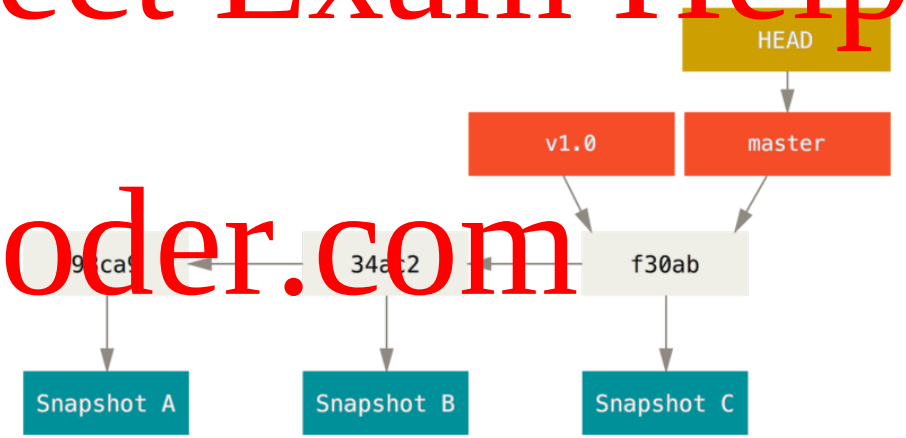


Figure 1: A branch and its commit history [1]

Example Git Workflows

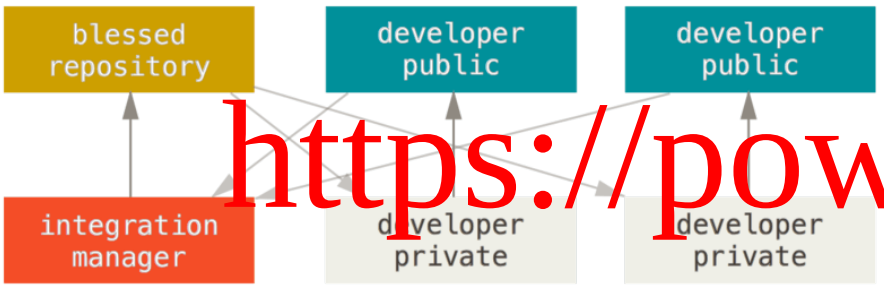


Figure: Integration-manager workflow [1]

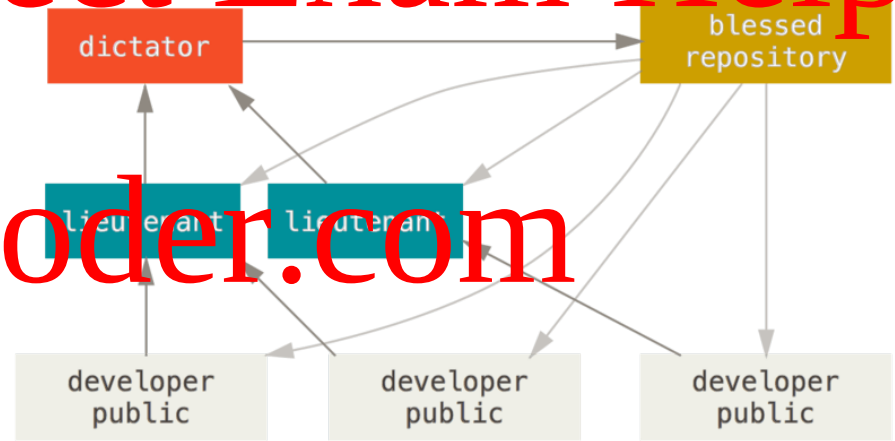


Figure: Benevolent dictator workflow [1]

Git Implements a Content-Addressable Filesystem

- A *content-addressable filesystem* is a way to store information so it can be retrieved based on its content, not its location
- What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content
- The same content will always have the same key

Exploring the .git directory

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Tree Objects

- Tree objects allow you to store filenames and groups of files together

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

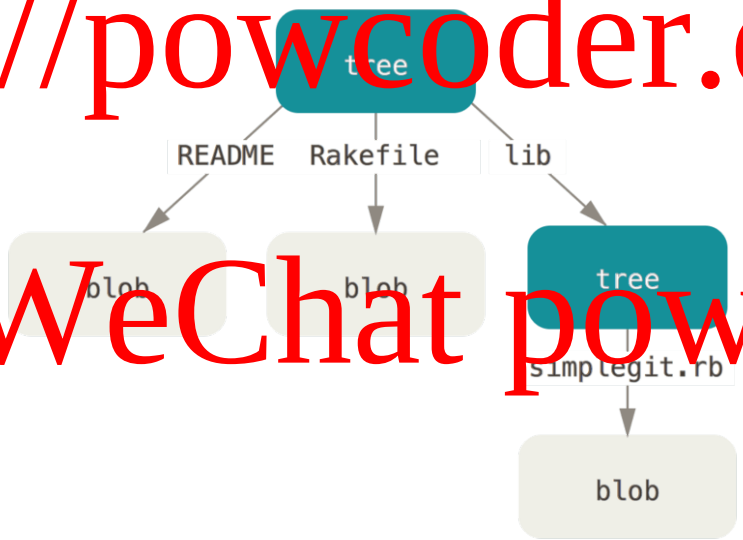



Figure: Simple version of the Git data model [1]

Commits Point to Trees and Add Metadata


```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

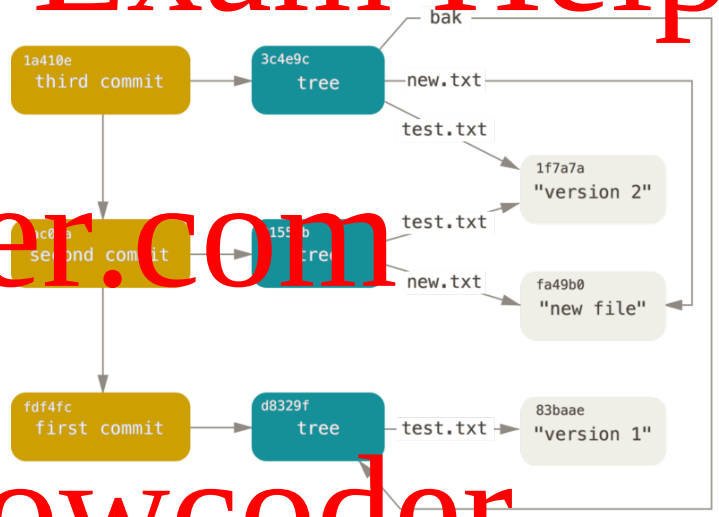



Figure: Visualizing commits and trees [1]

Git Packs Objects Into Pack Files

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/7f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/8c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

Consequences of Git's Design Decisions

- No access-control mechanisms built into Git
- More entities to keep track of (modified, staging, local repo, remote repo) and tools are more complicated to use
- Because source control is snapshot-based, differences over time must be synthesized (a.k.a. guessed) by tool
- Entire change history must be downloaded by all developers. If change history is large (e.g. has large binary files) this can be significant.
  - If a large binary is accidentally added to and then removed from a Git repo, it will still be in the Git history and required to be downloaded by all clients
  - Extensions like Git LFS and VFS for Git have been created to help deal with this
- If two pieces of content ever have a SHA-1 collision, Git will not be able to distinguish between them
- Committers can irrevocably destroy the contents of a repository with commands like `git rebase` and `git push -f`

Table of Contents

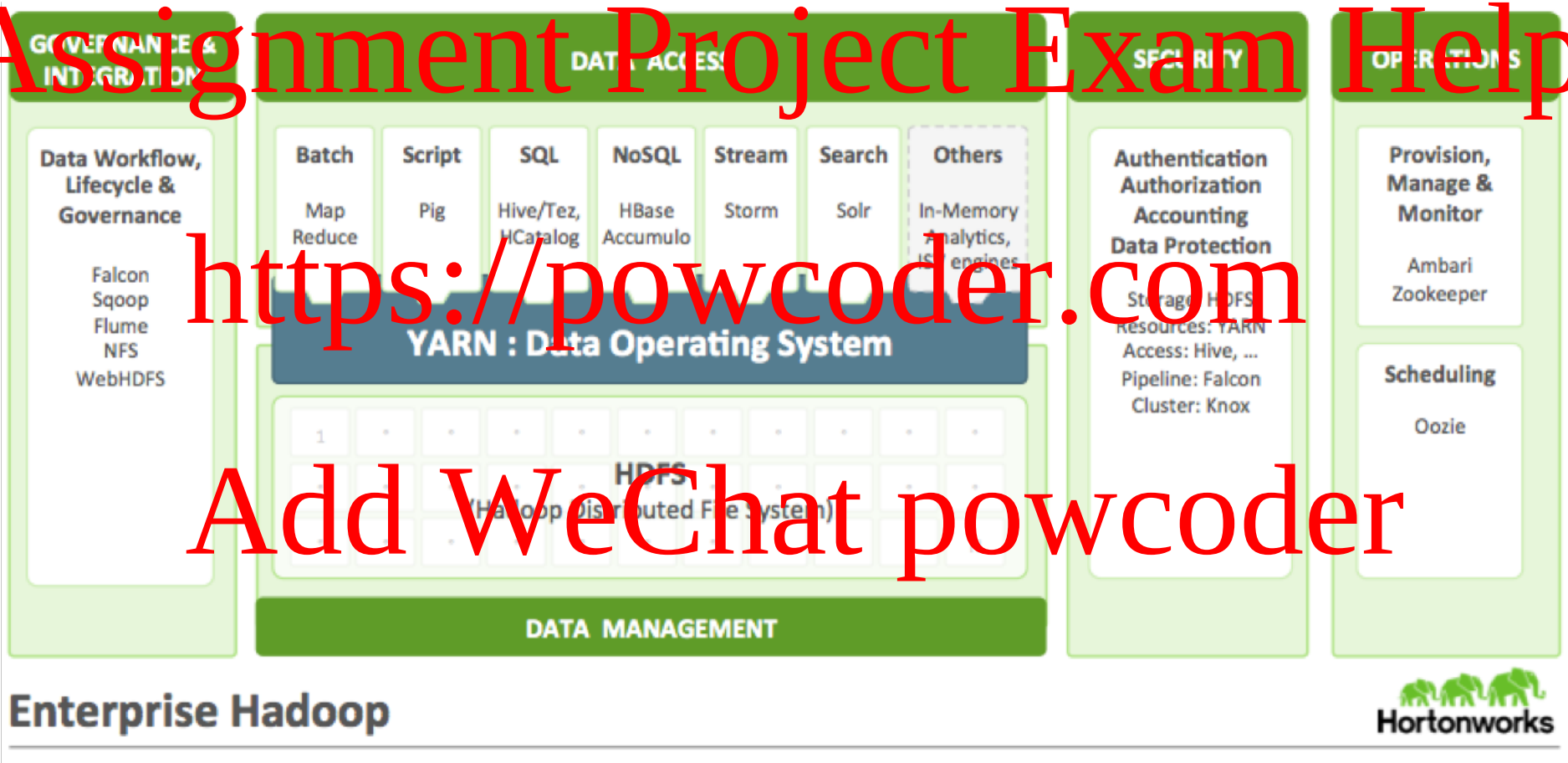
Hadoop

- The Hadoop project is one of the Apache Foundation's projects.
- Modeled after Google's MapReduce / GFS framework
- Implemented in Java
- Is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.
- It is designed to scale up from single server to thousands of machines, each offering local computation and storage. Rather than relying on hardware to deliver high availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.
- *Warning: This analysis is a few years old, so things might have changed since then.*

Hadoop Design Goals

- *Economical*
  - It is preferable to have more low-performance, low-cost hardware working in parallel than to have less high-performance, high-cost hardware
- *Reliable*
  - Failure becomes the norm
  - At some point, the number of discrete systems in a cluster will be greater than the mean time between failure (MTBF) for *any hardware platform*, no matter how reliable, so fault tolerance must be built into the controlling software.

Hadoop Ecosystem



Hadoop Subsystems We'll Talk About Today

- *Hadoop Distributed File System (HDFS)* – A distributed file system that provides high-throughput access to application data
- *Hadoop YARN* – A framework for job scheduling and cluster resource management
- *Hadoop Map-Reduce* – A YARN-based system for parallel processing of large data sets
- *Hadoop Common* – The common utilities that support the other Hadoop modules

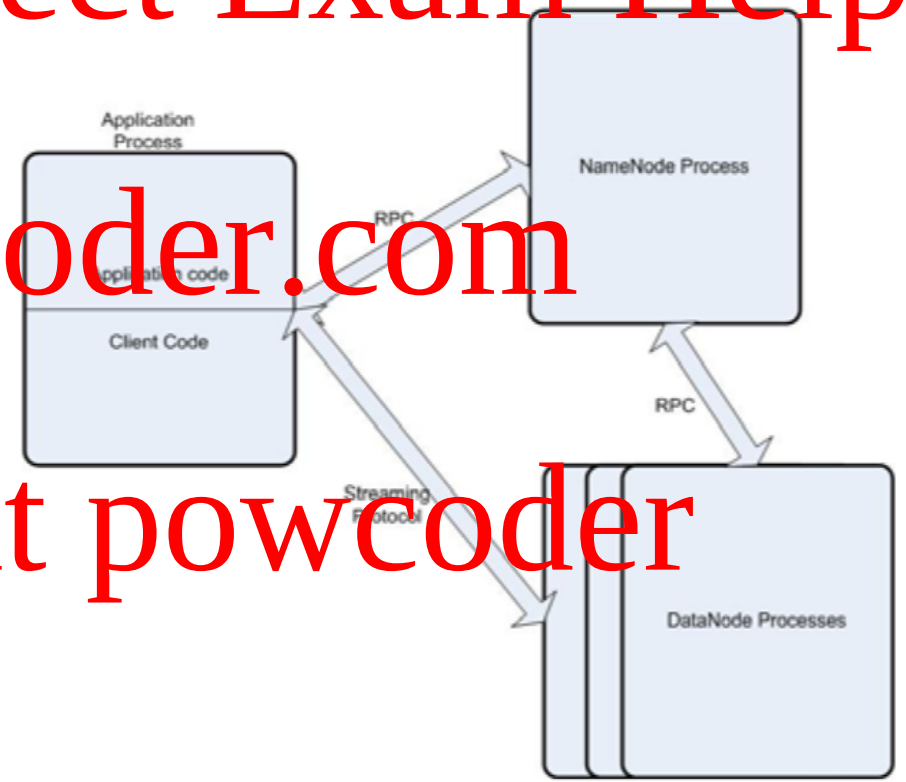
Hadoop's Main Architectural Style

- The *master-slave style* is dominant across all Hadoop's subsystems.
- In this architecture, slaves provide replicated services to the master, and the master selects a particular result among slaves by certain selection strategies. The slaves may perform the same functional task by different algorithms and methods or by a totally different functionality.

Table of Contents

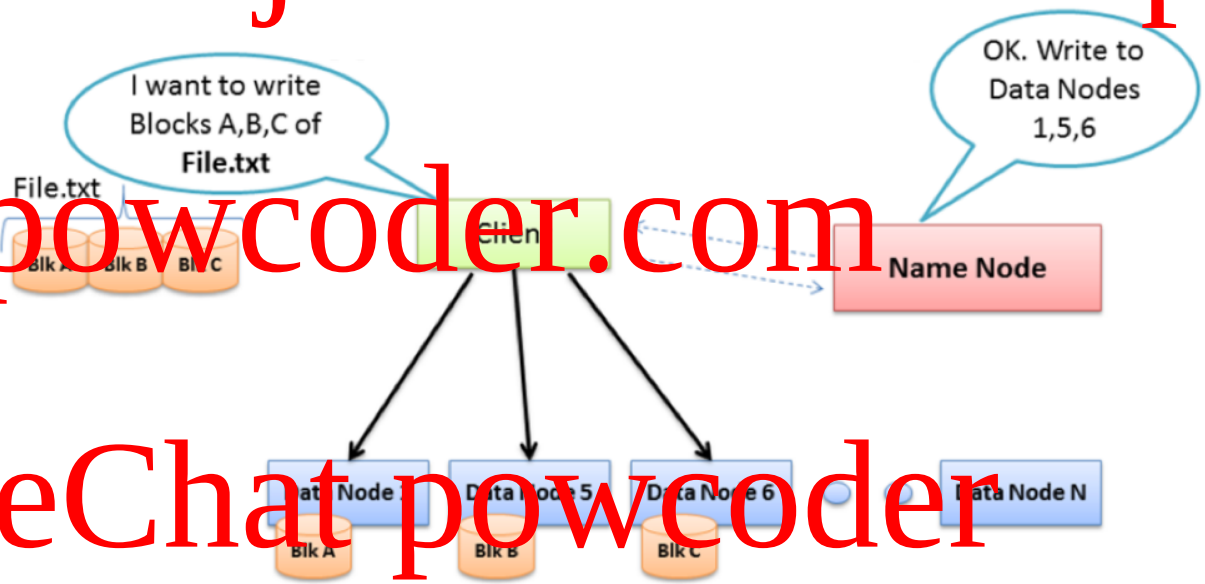
HDFS Architecture

- *NameNode*: A master server that manages the file system namespace and regulates access to files by clients.
- *DataNodes*: Stores the actual data in HDFS files. Usually one per node in the cluster.



Writing Files to HDFS

1. Client consults Name Node
2. Client writes block directly to one Data Node
3. Data Nodes replicates block
4. Cycle repeats for next block

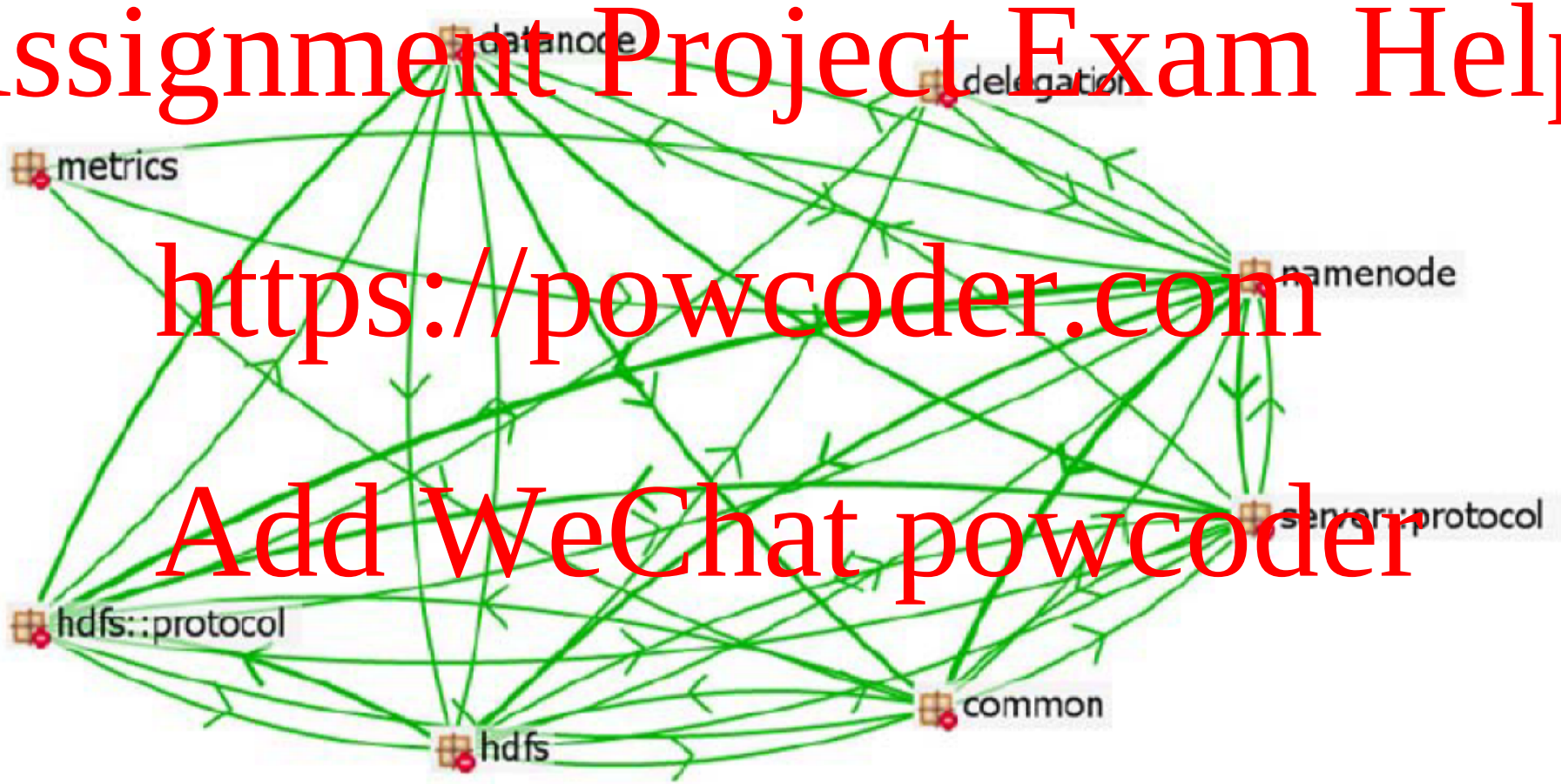


HDFS Architecture in Details

Code Structure:

- The library used by the client to communicate with the NameNode and the DataNodes.
- The protocols used for the client communication
- The NameNode code
- The DataNode code
- The protocols used for communication between the NameNode and the DataNodes.
- In addition, there are a few other important directories containing functionality that the HDFS code uses, such as Hadoop Common.

HDFS Code – A "Big Ball of Mud?"



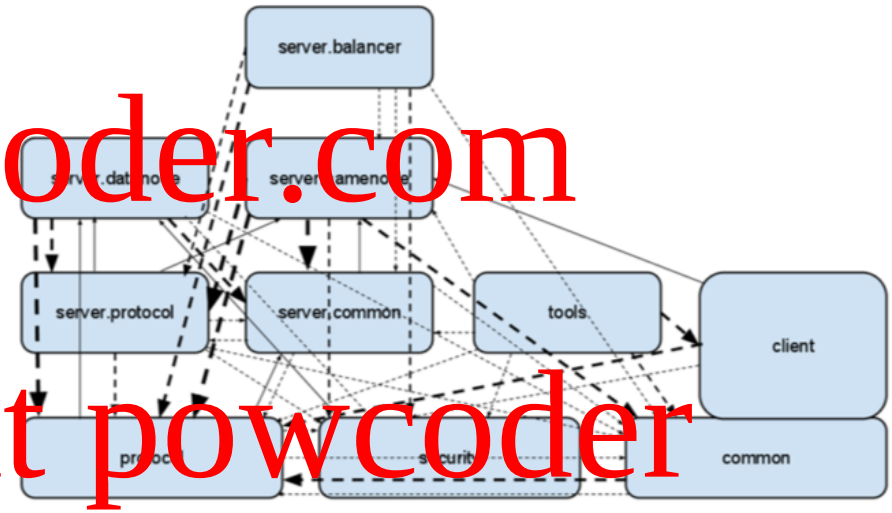
HDFS Architectural Issues

HDFS does not have any interesting layering.

- For example, since its portability concerns are, by and large, addressed by the technique of "implement in Java".
- The governing architectural pattern in HDFS is a master-slave style, which is a run-time structure.
- And modifiability, while important, has been addressed simply by keeping the code base at a relatively modest size and by having a significant number of committers spending considerable time learning and mastering this code base.

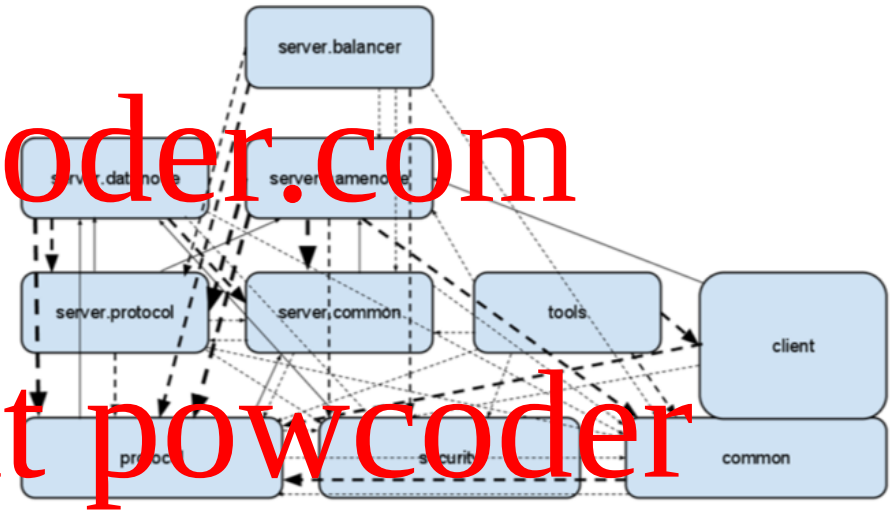
HDFS Module View

- *Protocol* is used in communication between the client and the namenode and datanode. It describes the messages used between these servers.

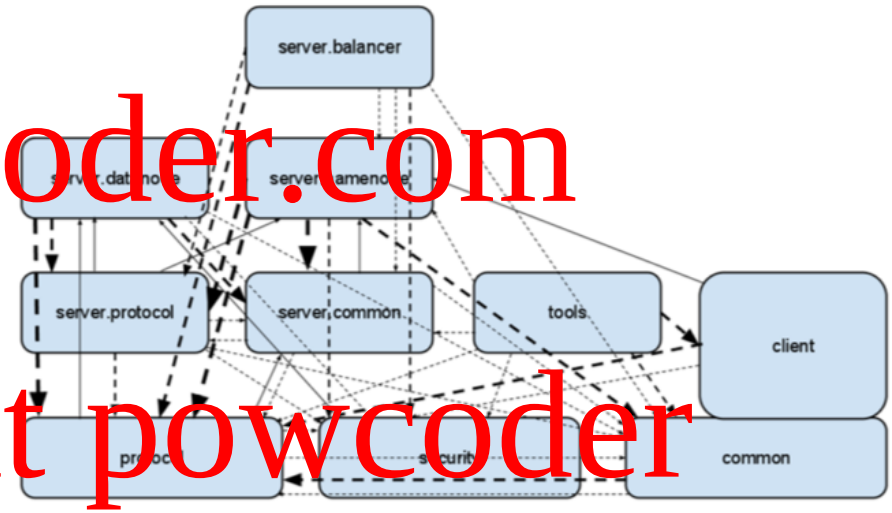


HDFS Module View

- *Security* is used in authenticating access to the files. The security is based on token-based authentication, where the namenode server controls the distribution of access tokens.

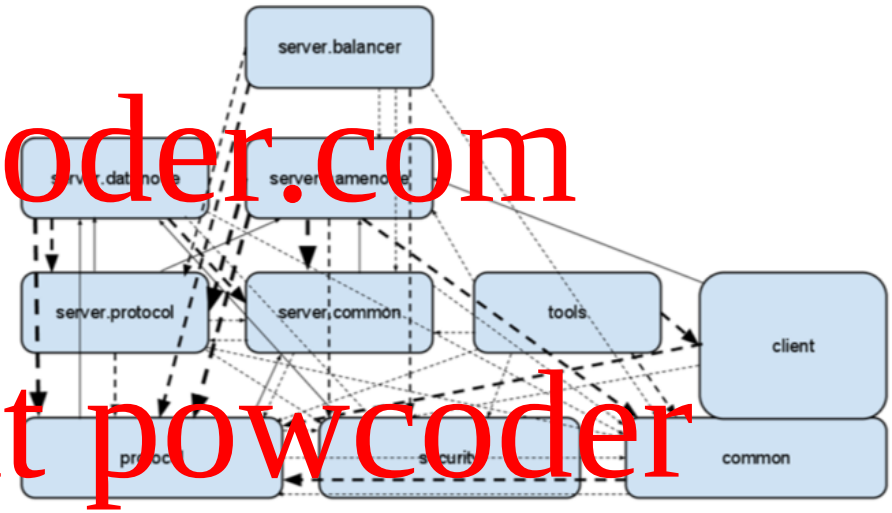


- *server.protocol* defines the communication between namenode and datanode, and between namenode and balancer.



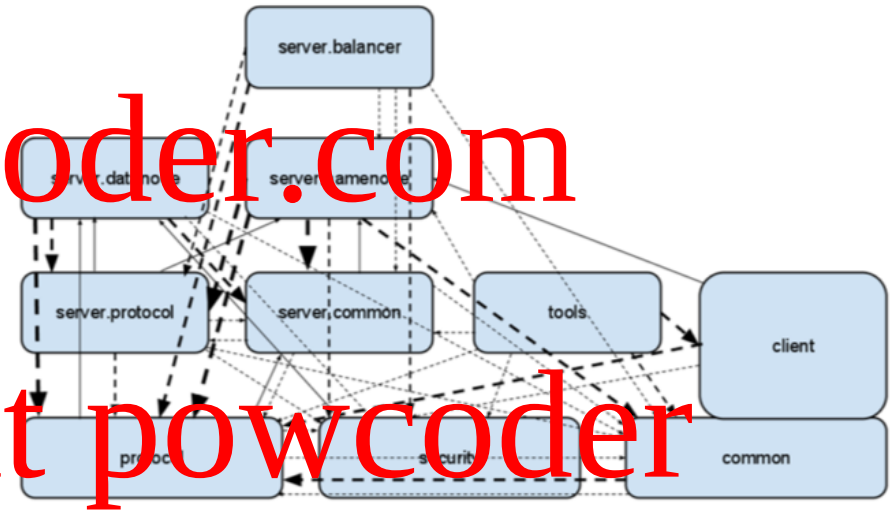
HDFS Module View

- *server.common* contains utilities that are used by the namenode, datanode and balancer. Examples are classes containing server-wide constants, utilities, and other logic that is shared among the servers.



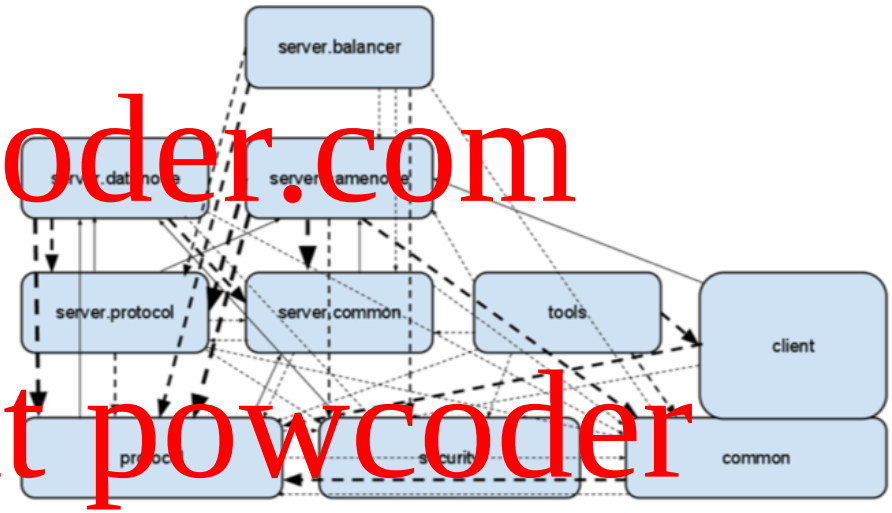
HDFS Module View

- *client* contains the logic to access the file system from a user's computer. It interfaces with the datanode and namenode servers using the protocol module.



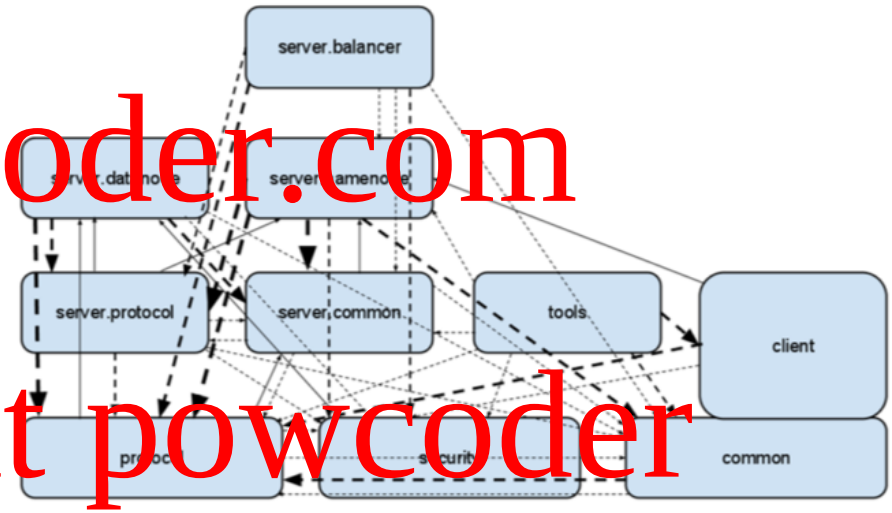
HDFS Module View

- *datanode* is responsible for storing the actual blocks of filesystem data. It receives instructions on which blocks to store from the namenode. It also services the client directly to stream file block contents.



HDFS Module View

- *namenode* is responsible for authorizing the user, storing a mapping from filenames to data blocks, and it knows which blocks of data are stored where.



HDFS Module View

- *tools* can be used to administer the filesystem, and also contains debugging code.

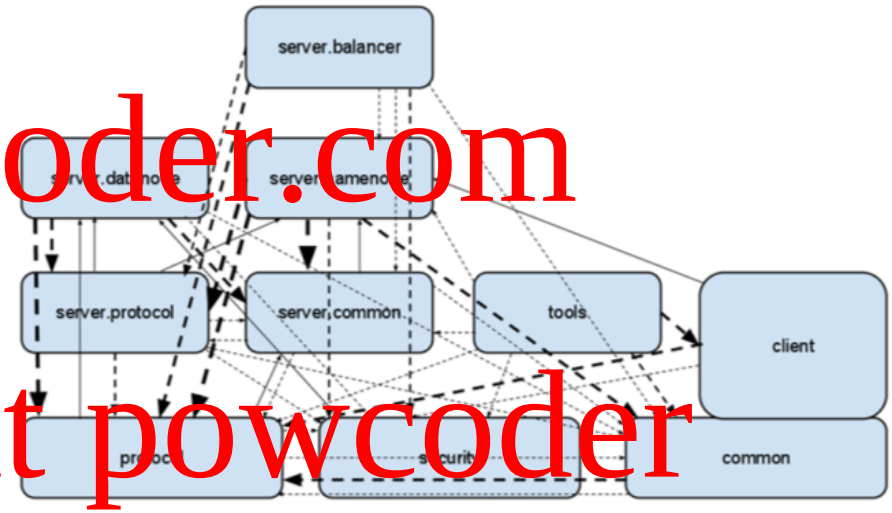
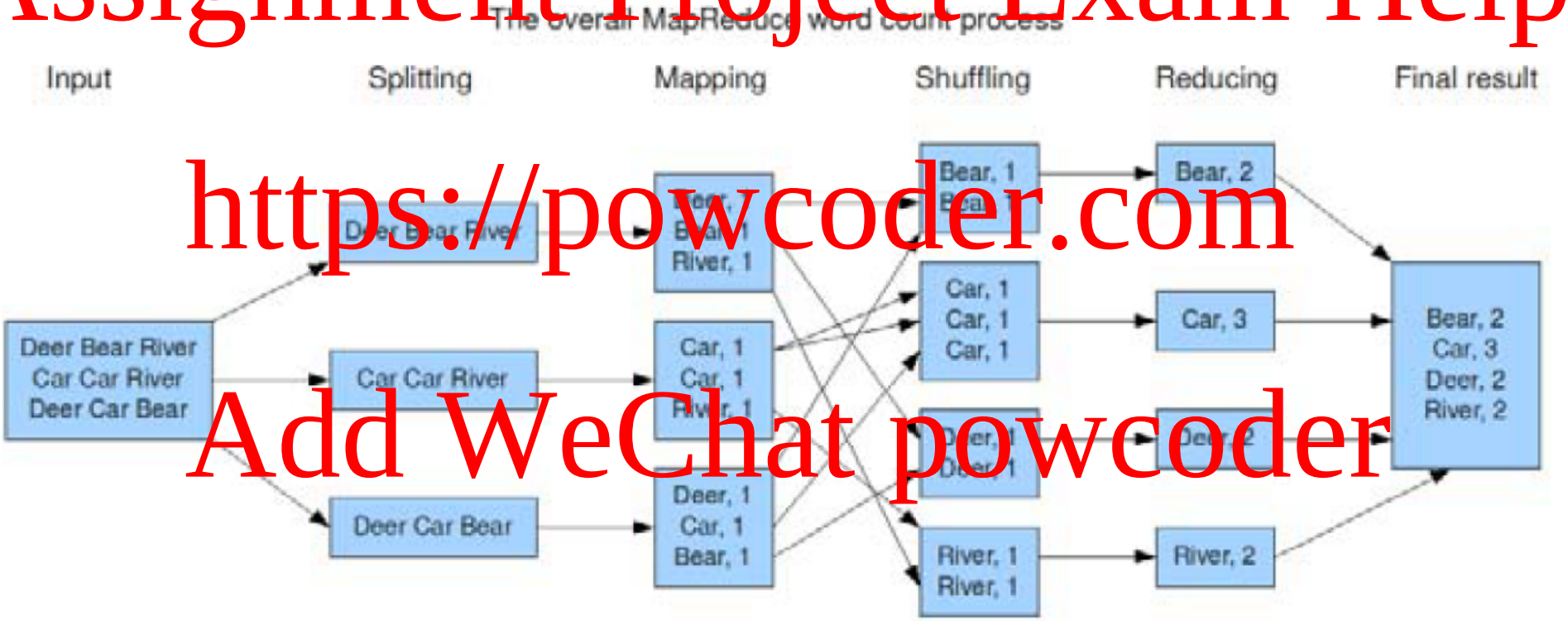


Table of Contents

Hadoop Map-Reduce Architecture

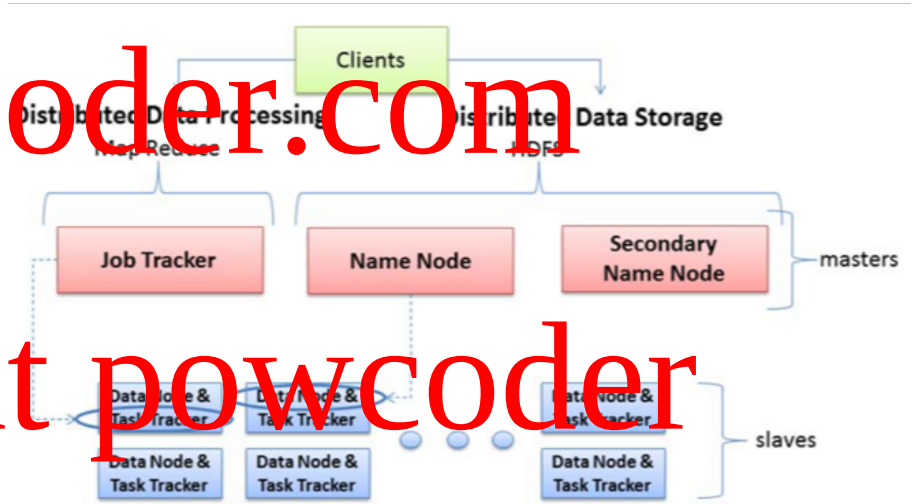


Hadoop Map-Reduce Architecture

- *JobTracker*: Oversees and coordinates the parallel processing of data using MapReduce
- *TaskTracker*: Does the actual work given by JobTracker – it does either map or reduce

Combined Architectural View

1. Client machine submits the Map Reduce job to the Job Tracker, asking "How many times does Ca occur in File.txt"
2. The Job Tracker consults the Name Node to learn which Data Nodes have blocks of File.txt
3. The Job Tracker then provides the Task Tracker running on those nodes with the Java code required to execute the Map computation on their local data.
4. The Task Tracker starts a Map task and monitors the tasks progress. The Task Tracker provides heartbeats and task status back to the Job Tracker.



Another Architectural View

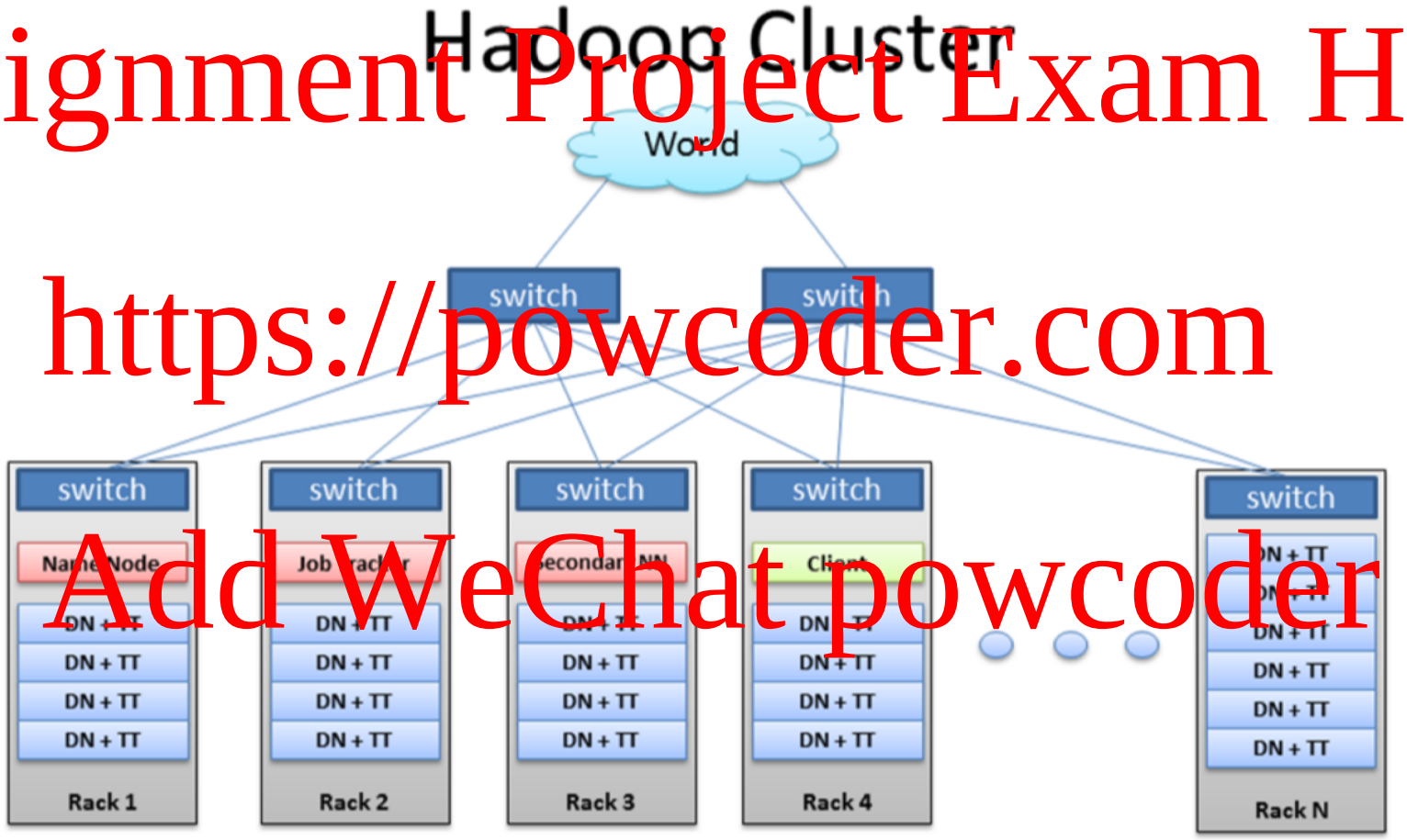
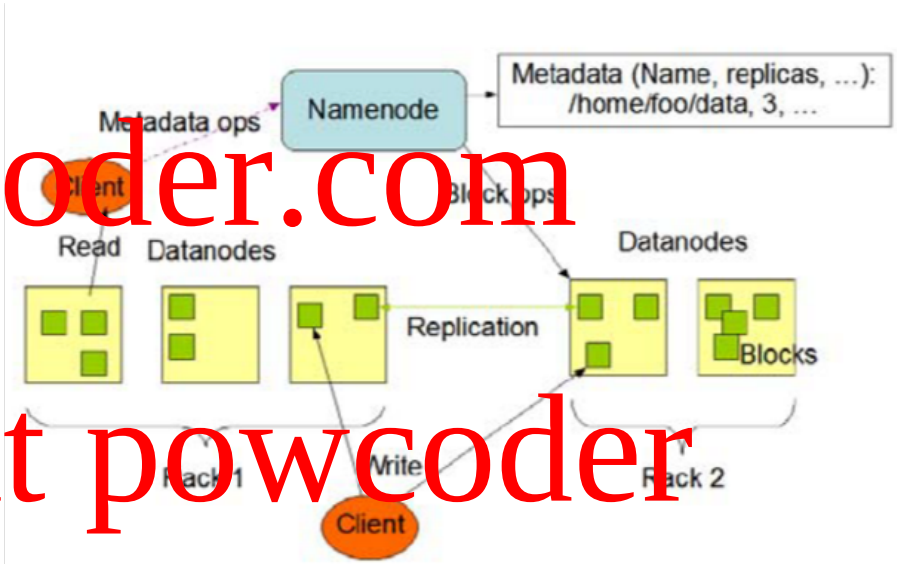


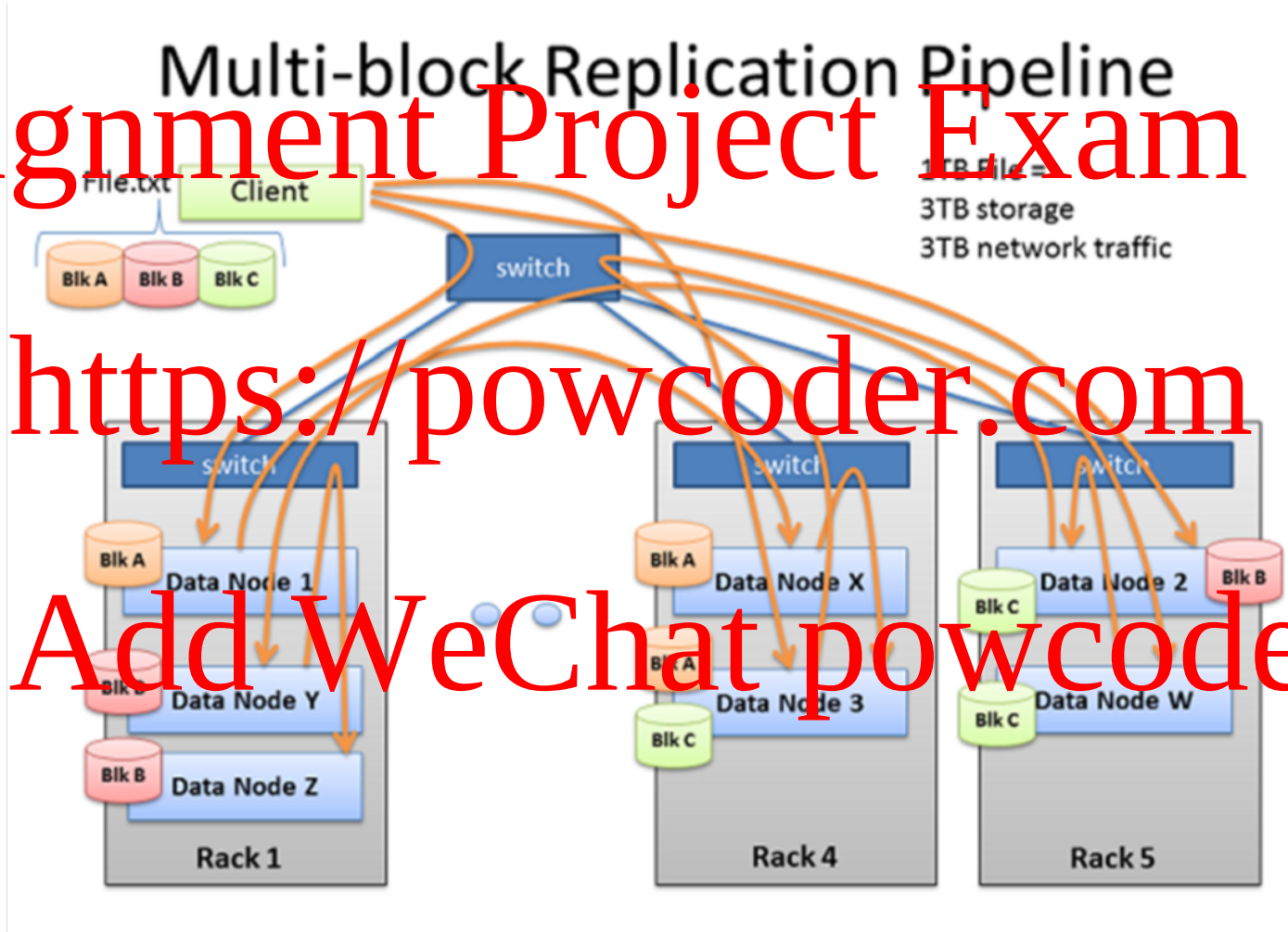
Table of Contents

Availability in HDFS

- *Availability* is managed by maintaining multiple replicas of each block in an HDFS file, recognizing failure in a DataNode or corruption of a block, and having mechanisms to replace a failed DataNode or a corrupt block.



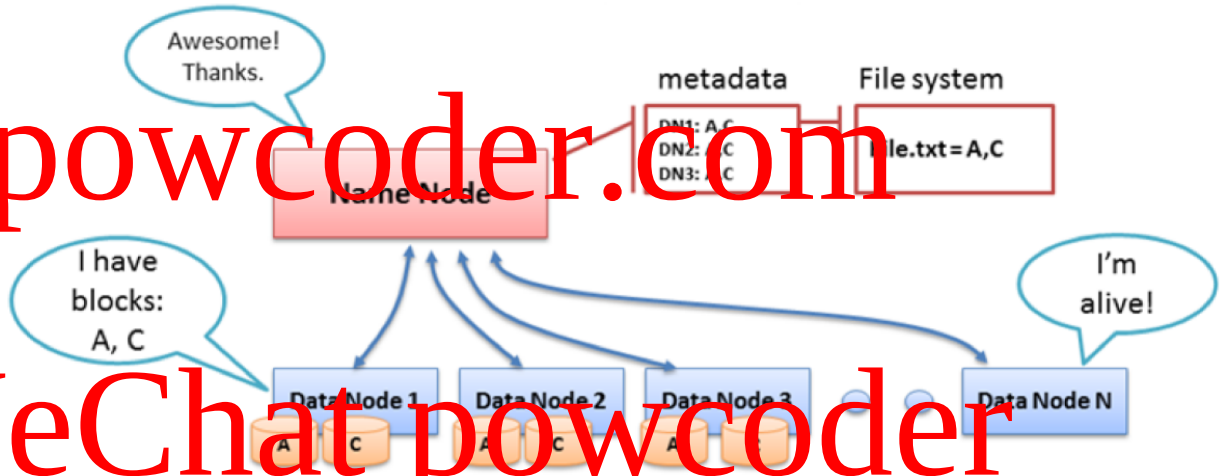
Availability in HDFS



Availability in HDFS

Detecting Failure:

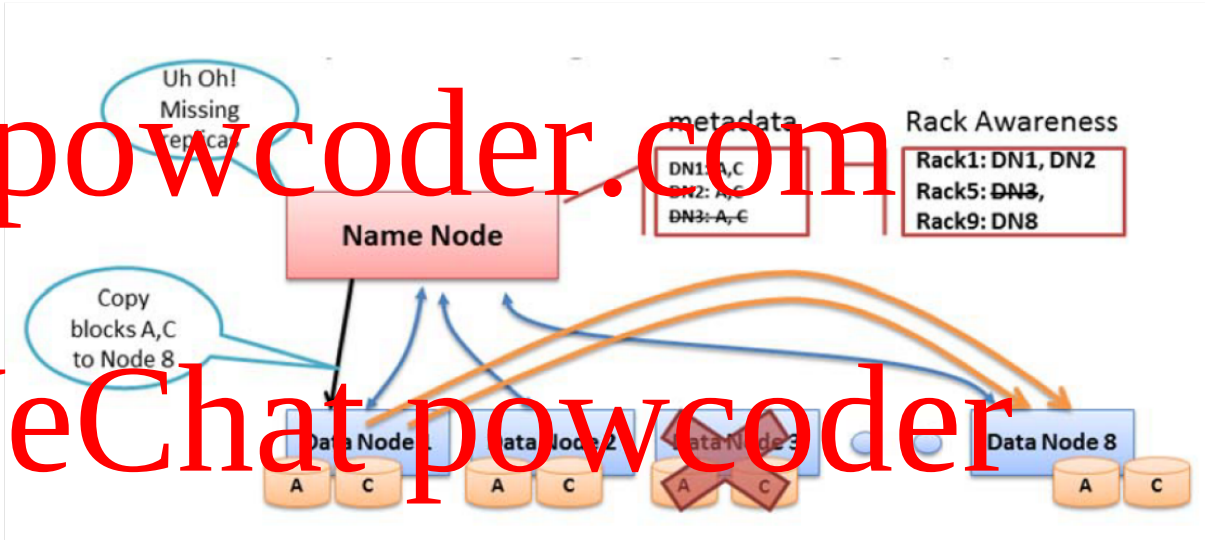
- Data node sends heartbeats
- Every 10th heartbeat is a block report
- Name node builds metadata from block reports every 3 seconds
- If name node is down, HDFS is down



Availability in HDFS

Re-replicating the Missing Replicas:

- Missing heartbeats signify lost nodes
- Name node consults metadata, finds affected data
- Name node consults Rack Awareness script
- Name node tells a data node to re-replicate



Single Point of Failure (as of Hadoop 2.0.0)

- In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted the NameNode.
- Planned maintenance events such as software or hardware upgrades on the NameNode machine would result in windows of cluster downtime
- Secondary Name Node not a hot standby for the Name Node
- Connects to Name Node every hour for Housekeeping, backup of Name Node metadata
- Saved metadata can rebuild a failed Name Node

Single Point of Failure

- In a typical HA cluster, two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an Active state, and the other is in a Standby state.
- The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary.
- To keep the state synchronized, the current implementation requires that the two nodes both have access to a directory on a shared storage device (eg an NFS mount from a NAS). This shared storage device is a single point of failure.
- This restriction will likely be relaxed in future versions.

HDFS Architectural Issues (as of Hadoop 2.0.0)

- When any namespace modification is performed by the Active node, it durably logs a record of the modification to an edit log file stored in the shared directory. The Standby node is constantly watching this directory for edits, and as it sees the edits, it applies them to its own namespace.
- For fast failover, the Standby node must have up-to-date information regarding the location of blocks in the cluster. Therefore the DataNodes are configured with the location of both NameNodes, and send block location information and heartbeats to both.
- Only manual failover is supported. HA NameNodes are incapable of automatically detecting a failure of the Active NameNode, and instead rely on the operator to manually initiate a failover. Automatic failure detection and initiation of a failover will be implemented in future versions.

Table of Contents

Hadoop Performance

Different Scheduling Mechanisms:

- *FIFO*: Under Hadoop's default FIFO scheduler as soon as a job is sent to Hadoop for execution, the JobTracker will assign as many TaskTrackers as necessary to process that job.
  - Express Checkout for smaller Jobs
- *FAIR Scheduler*: Multiple jobs, each having a Pool. Each Pool gets a guaranteed number of task slots (map/reduce)
  - "Fair sharing," such that each job gets roughly an equal amount of compute resource after exceeding the pool size.
- *Capacity Scheduler*: shares similar goals with the Fair Scheduler. The Capacity Scheduler works on queues rather than pools.
  - Within a queue, jobs have priority
  - Queues are guaranteed a fraction of the capacity of the grid
  - Each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them

Hadoop Load Balancing

1. Acquire neighborhood details:
   - When the load increases in a DataNode to the threshold level, it sends a request to the NameNode.
   - The NameNode has information about the load levels of the specific DataNode's nearest neighbors.
   - Loads are compared by the NameNode and then the details about the free-est neighbor nodes are sent to the specific DataNode.
2. DataNodes go to work:
   - Each DataNode compares its own load amount with the sum of the load amount of its nearest neighbors.
   - If a DataNode's load level is greater than the sum of its neighbors, then load-destination nodes (direct neighbors AND other nodes) will be chosen at random.
   - Load requests are then sent to the destination nodes.
3. Request is received:
   - Buffers are maintained at every node to received load requests.
   - A message passing interface (MPI) manages this buffer.
   - A main thread will listen to the buffered queue and will service the requests it receives.
   - The nodes enter the load-balancing-execution phase.

Table of Contents

HDFS Security

- Goal: Keep data in HDFS secure from unauthorized access.
- Corollary:
  - Users must be authenticated
  - Since Map/Reduce run applications as user, they must authenticate the users.
  - Since servers HDFS or Map/Reduce are entrusted with user credentials, they must also be authenticated
  - Kerberos is the key authentication system
  - Security on/off option

HDFS Security

Prevent unauthorized HDFS access:

- All HDFS clients must be authenticated.
  - Including tasks running as part of MapReduce jobs
  - Submitted jobs
- Users must also authenticate servers
  - Otherwise fraudulent servers could steal credentials
- Integrate Hadoop with Kerberos
  - Provides well tested open source distributed authentication system

HDFS Security

Authentication: Determining the user

- Hadoop 0.20 completely trusted the user
  - User passes their username and groups over wire
- We need it on both RPC and Web UI

Authorization: What can that user do?

- HDFS had owners, groups and permissions since 0.16.
- Map/Reduce had nothing in 0.20

HDFS Security

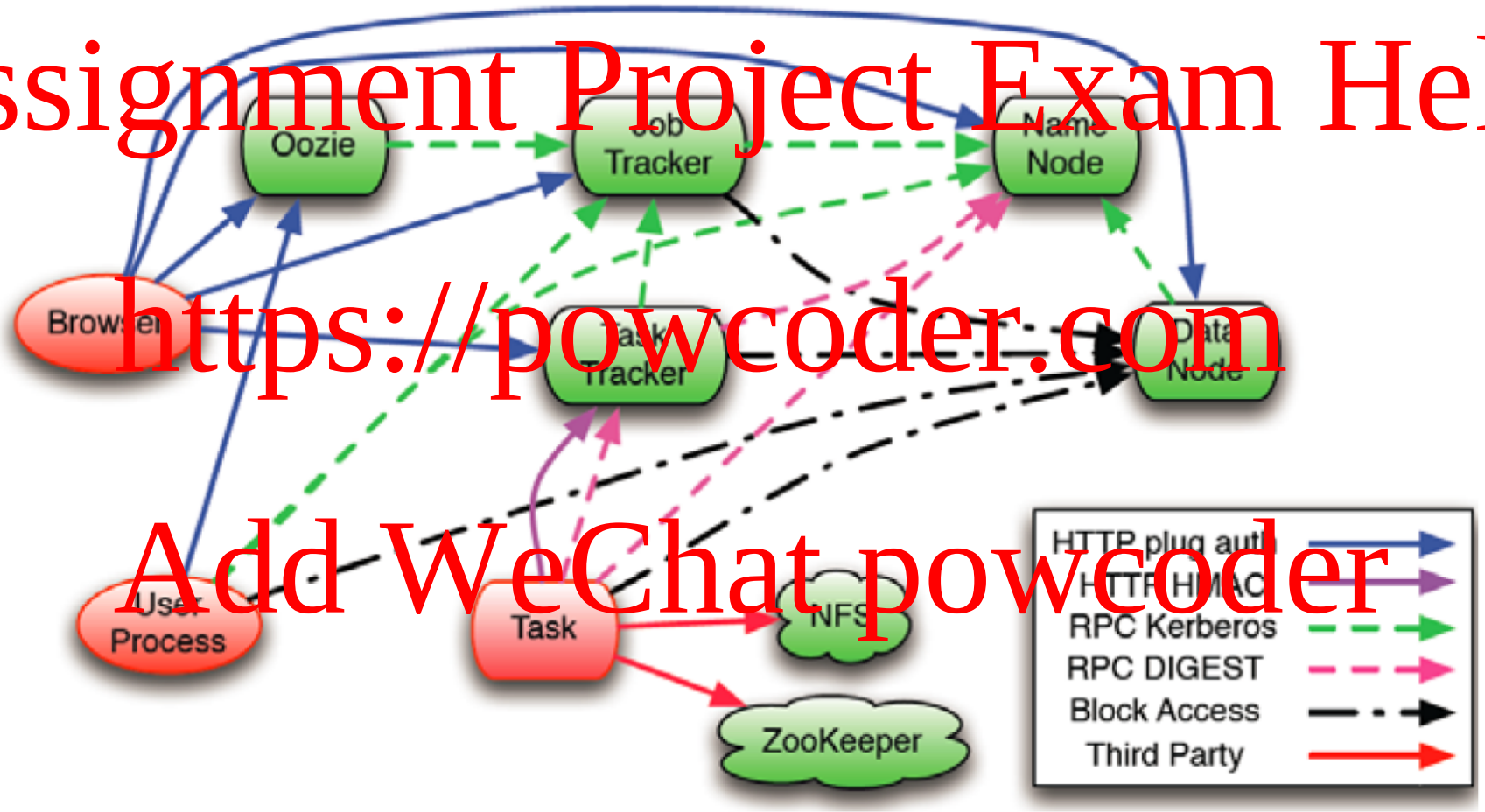


Table of Contents

https://www.infoq.com/presentations/reddit-architecture-evolution

Table of Contents

Wrap-Up

- Homework 5 due Thursday, November 19 at 5:30PM
- Quiz 5 due Thursday, November 12 at 5:30PM
- Next week is last lecture. We will review Quiz 4, Quiz 5, and Homework 4.
- Final exam in two weeks. More details next week.

References

[1] Chacon, S., and Straub, B.
Pro git: Everything you need to know about Git, second ed.
Apress, 2014.

[2] History, T.-B.
Version control before git with cvs.

[3] Wikipedia.
Git.