云计算技术与工程

大数据存储技术

徐敏贤

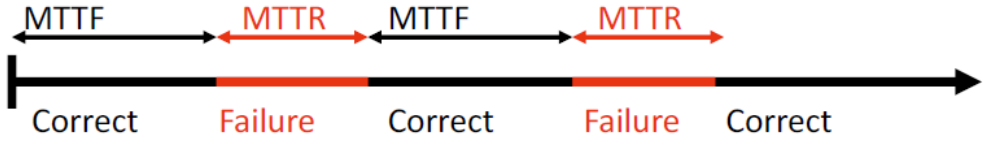
副研究员
云计算研究中心
中国科学院深圳先进技术研究院
http://www.minxianxu.info/vcc

暮色苍茫看劲松，乱云飞渡仍从容。天生一个仙人洞，无限风光在险峰。
——毛泽东

Review

- 1. 系统的Reliability通常可以使用哪几个指标度量?

- Reliability: measure of continuous “service”
  - MTTF: mean time to failure
- Time to produce first incorrect output
  - MTTR: mean time to repair
- Time to detect and repair a failure
  - MTBF = mean time between failures = MTTF+MTTR
- Failure in time (FIT) = failures per billion hours of operation = 10^9/MTTF
  - E.g., MTTF = 1,000,000 hours →1,000 FIT



Review

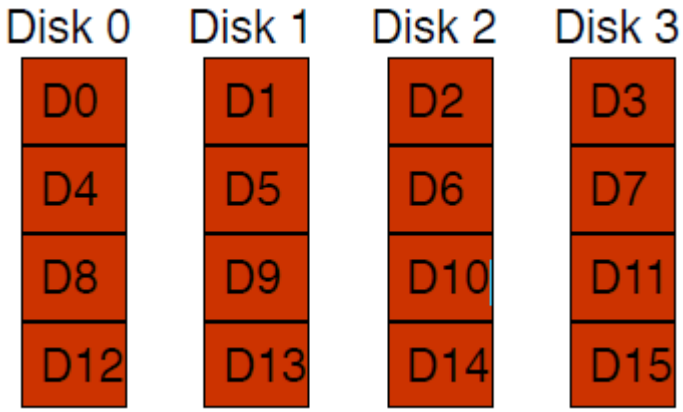
- 2. 哪些方法可以提升MTTF和MTTR?

- High-level steps
  - Units built in a way that simulates factory methods
  - All components evaluated: electrical, mechanical, SW bundles, firmware, system
  - Interoperability
  - Failure diagnostics and iteration with design team
  - Potential beta customer testing
- Extensive testing
  - Accelerated thermal lifetime testing (-60C to 90C)
  - Accelerated vibration testing
  - Manufacturing verification – accelerated stress audit
  - Reliability of user interface and full rack configurations
  - Static discharge, repetitive mechanical joints, …
  - Dust chamber: simulate dust buildup
  - Environmental testing – model shipping stresses
  - Acoustic emissions and EMI standards
  - FCC approval (US), CE approval (Europe)
  - Power fluctuations and noise – semi-anachoic chamber
  - On-site data center testing – TPC benchmarking

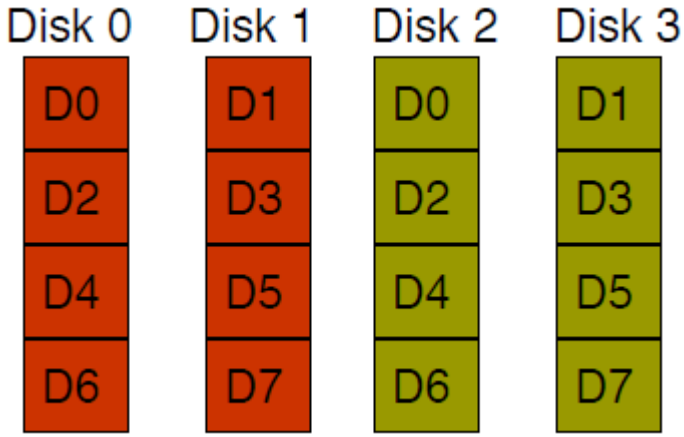
Review

- 3. RAID 0和RAID 1/1+0分别有什么优缺点?

- RAID 0: Non-redundancy (sharding)
  - Best write performance
  - Not necessarily the best read latency!
  - Worst reliability



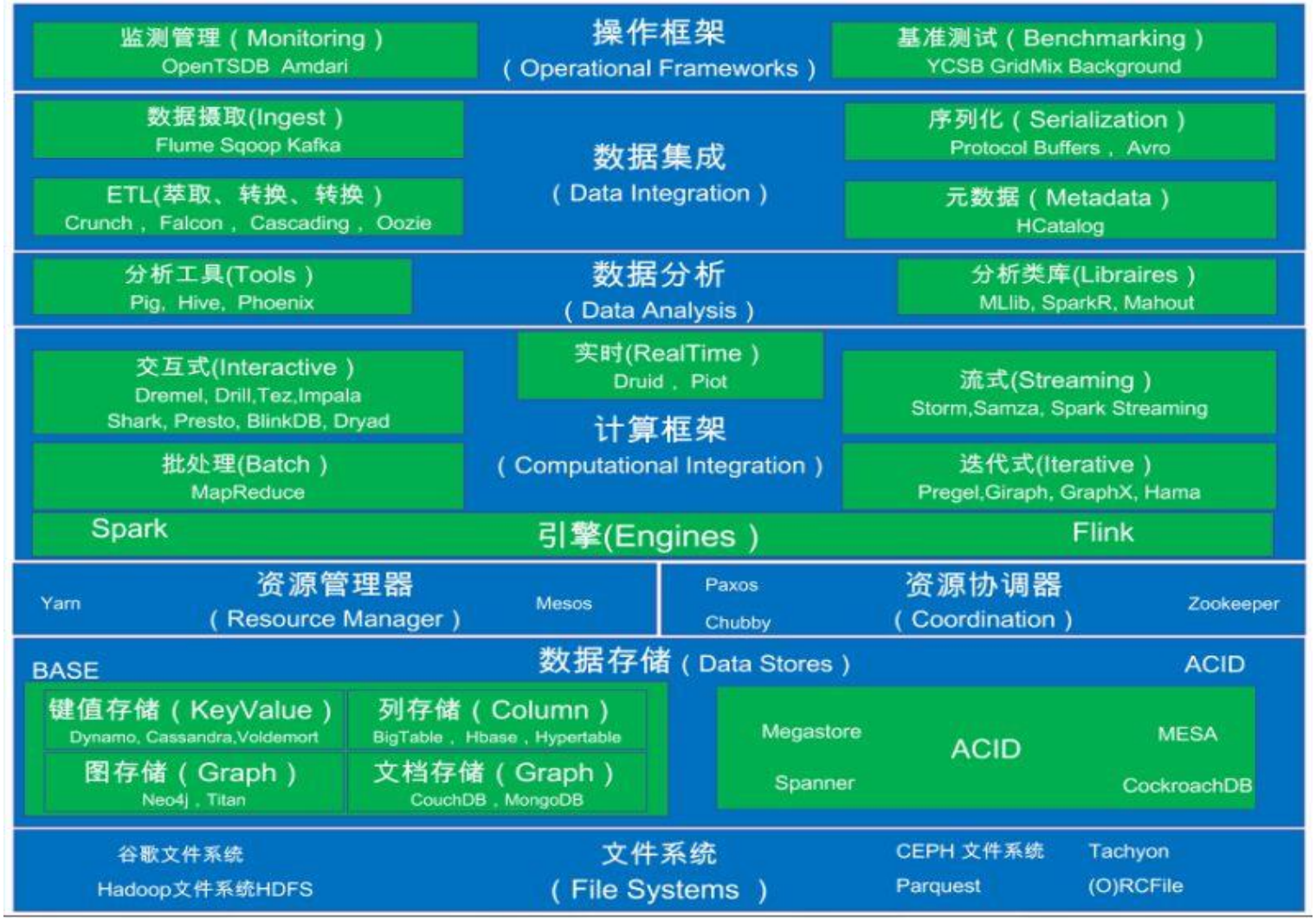
- RAID 1/1+0: Mirroring (shadowing)
  - Half the capacity
  - Faster read latency: schedule the disk with the lowest queuing, seek, and rotational delays

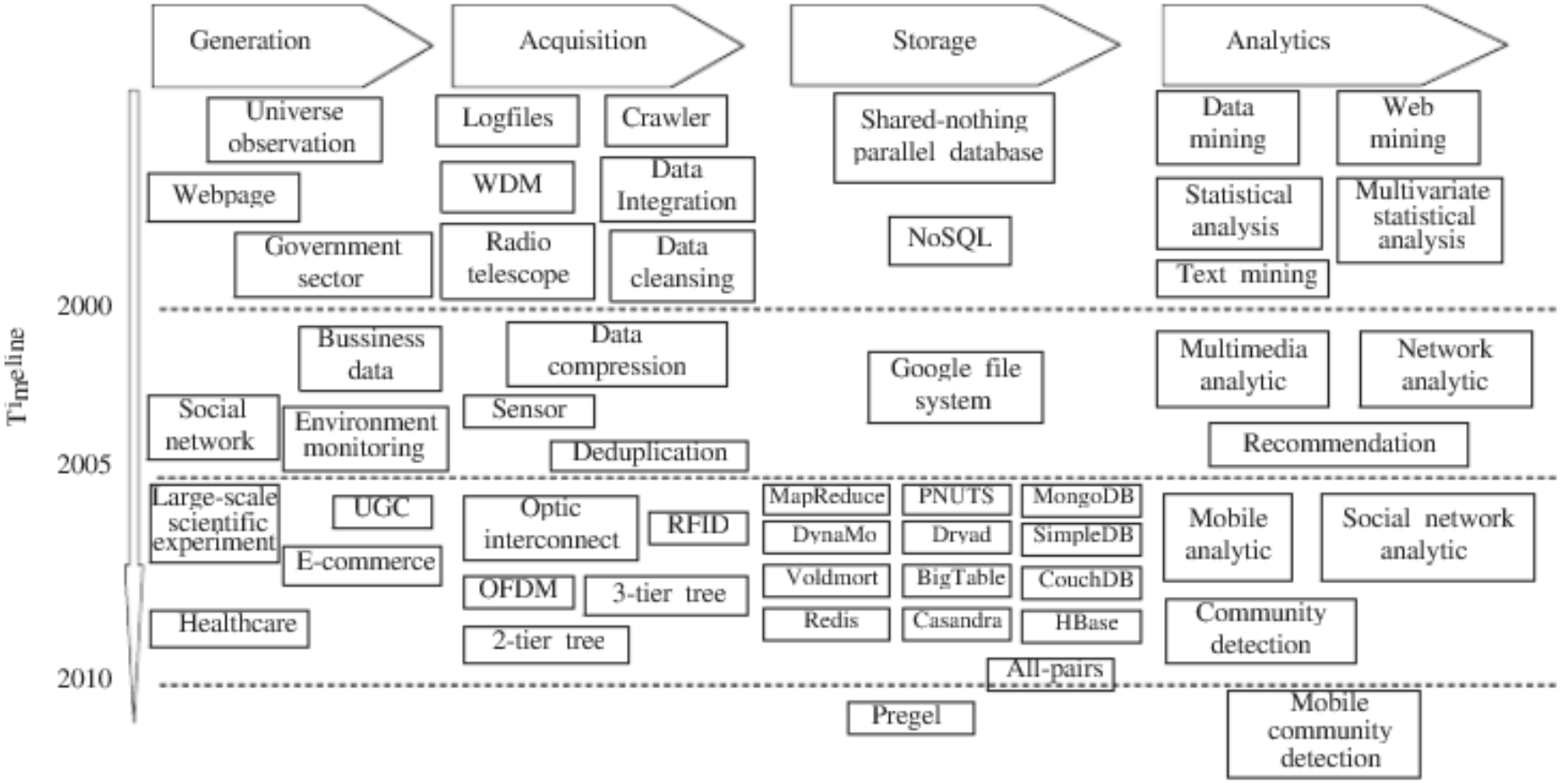


目录 Contents

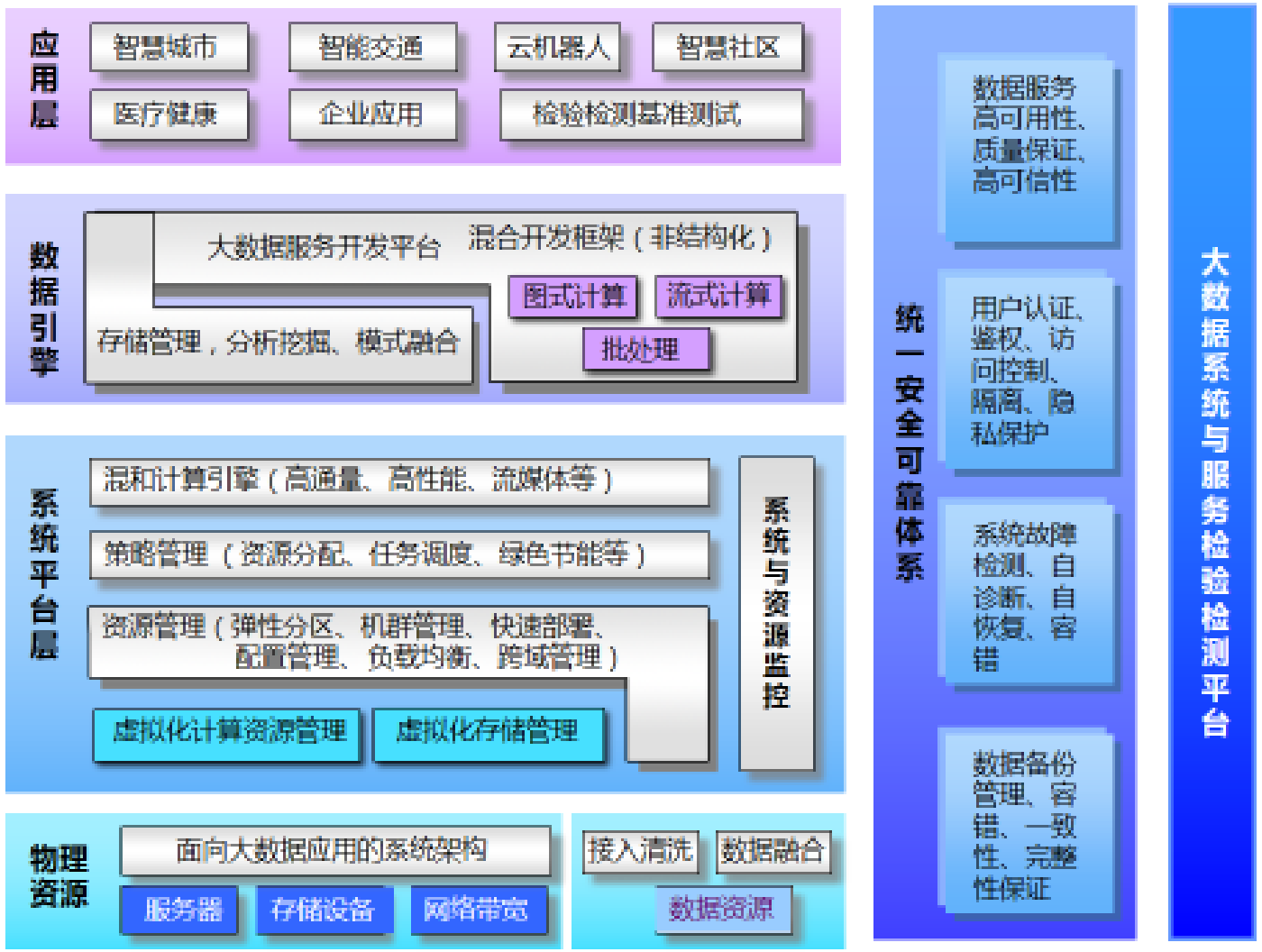
注：本章部分内容由王洋教授提供

大数据处理关键架构


监测管理（Monitoring）
OpenTSDB Amdari
操作框架
（Operational Frameworks）
基准测试（Benchmarking）
YCSB GridMix Background
数据摄取(Ingest）
Flume Sqoop Kafka
序列化（Serialization）
Protocol Buffers，Avro
数据集成
（Data Integration）
ETL(萃取、转换、转换）
Crunch，Falcon，Cascading，Oozie
元数据（Metadata）
HCatalog
分析工具(Tools）
Pig，Hive，Phoenix
数据分析
（Data Analysis）
分析类库(Libraires）
MLlib, SparkR, Mahout
交互式(Interactive）
Dremel, Drill,Tez,Impala
Shark, Presto, BlinkDB, Dryad
实时(RealTime）
Druid，Piot
计算框架
（Computational Integration）
流式(Streaming）
Storm,Samza, Spark Streaming
批处理(Batch）
MapReduce
迭代式(Iterative）
Pregel,Giraph, GraphX, Hama
Spark
引擎(Engines）
Flink
Yarn
资源管理器
（Resource Manager）
Mesos
Paxos
Chubby
资源协调器
（Coordination）
Zookeeper
BASE
数据存储（Data Stores）
ACID
键值存储（KeyValue）
Dynamo, Cassandra,Voldemort
列存储（Column）
BigTable，Hbase，Hypertable
图存储（Graph）
Neo4j，Titan
文档存储（Graph）
CouchDB，MongoDB
Megastore
Spanner
ACID
MESA
CockroachDB
谷歌文件系统
Hadoop文件系统HDFS
文件系统
（File Systems）
CEPH 文件系统
Parquest
Tachyon
(O)RCFile


Generation
Acquisition
Storage
Analytics
Timeline
2000
2005
2010
Universe observation
Webpage
Government sector
Logfiles
Crawler
WDM
Data Integration
Radio telescope
Data cleansing
Shared-nothing parallel database
NoSQL
Data mining
Web mining
Statistical analysis
Multivariate statistical analysis
Text mining
Bussiness data
Data compression
Social network
Environment monitoring
Sensor
Deduplication
Google file system
Multimedia analytic
Network analytic
Recommendation
Large-scale scientific experiment
UGC
E-commerce
Optic interconnect
RFID
OFDM
3-tier tree
2-tier tree
Healthcare
MapReduce
DynaMo
Voldmort
Redis
PNUTS
Dryad
BigTable
Casandra
MongoDB
SimpleDB
CouchDB
HBase
All-pairs
Pregel
Mobile analytic
Social network analytic
Community detection
Mobile community detection

大数据框架研讨


应用层
智慧城市
智能交通
云机器人
智慧社区
医疗健康
企业应用
检验检测基准测试
数据引擎
大数据服务开发平台
混合开发框架（非结构化）
图式计算
流式计算
批处理
存储管理，分析挖掘、模式融合
系统平台层
混和计算引擎（高通量、高性能、流媒体等）
策略管理（资源分配、任务调度、绿色节能等）
资源管理（弹性分区、机群管理、快速部署、
配置管理、负载均衡、跨域管理）
虚拟化计算资源管理
虚拟化存储管理
系统与资源监控
物理资源
面向大数据应用的系统架构
服务器
存储设备
网络带宽
接入清洗
数据融合
数据资源
统一安全可靠体系
数据服务高可用性、质量保证、高可信性
用户认证、鉴权、访问控制、隔离、隐私保护
系统故障检测、自诊断、自恢复、容错
数据备份管理、容错、一致性、完整性保证
大数据系统与服务检验检测平台

1. 文件系统层：存储管理、容错处理、高可扩展性、高可靠性和高可用性等特性。
2. 数据存储层：非结构化和半结构化数据，数据的表现形式各异，有基于键值（Key-Value）的，有基于文档（Document），还有基于列（Column）和图表（Graph）
3. 资源管理层：这一层是为了提高资源的高利用率和吞吐量，以到达高效的资源管理与调度目的。
4. 资源协调层： 在本层的系统，需要完成对资源的状态、分布式协调、一致性和资源锁实施管理。
5. 计算框架层：高度专用的框架包含其内，有流式的，交互式的，实时的，批处理和迭代图的
6. 数据分析层：数据分析(消费)工具和一些数据处理函数库。
7. 数据集成层：管理数据分析工作流中用到的各种适用工具，包括对元数据管理的工具。
8. 操作框架层：这一层提供可扩展的性能监测管理和基准测试框架。

文件系统层

基于磁盘存储的文件系统向基于内存计算的文件系统转变

1. Google文件系统（GFS）
2. Hadoop 文件系统 (HDFS)
3. Ceph文件系统: Linux系统下 PB 级分布式文件系统
4. Tachyon: 高容错的分布式内存文件系统
5. GlusterFS （无元服务器结构，多租户原型系统）
6. GPFS/pNFS/PVFS (OrangeFS)

文件系统中的一些技术

1.面向行存储 vs. 面向列存储（RCFile，Parquet， ORCFile）
2.压缩技术 （2014 SIGMOD综述）
3.纠删码技术 （James Plank综述）
4.元数据管理技术 (集成技术综述)
5.软件定义存储技术 （Toronto，MIT）
6.新型存储介质应用 （华中科技）

数据存储层

对一致性（consistency）要求的强弱不同，分布式数据存储策略，可分为ACID和BASE两大阵营。

•ACID：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。ACID中的一致性要求比较强，事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态

1. Megastore,
2. Spanner,
3. Mesa

•BASE：基本可用（Basically Available），软状态/柔性事务（Soft-state，即状态可以有一段时间的不同步），最终一致性（Eventual consistency）。

类型	部分代表	特点
列存储	Hbase Cassandra Hypertable	顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对针对某一列或者某几列的查询有非常大的IO优势。
文档存储	MongoDB CouchDB	文档存储一般用类似json的格式存储，存储的内容是文档型的。这样也就有有机会对某些字段建立索引，实现关系数据库的某些功能。
key-value存储	Tokyo Cabinet / Tyrant Berkeley DB MemcacheDB Redis	可以通过key快速查询到其value。一般来说，存储不管value的格式，照单全收。（Redis包含了其他功能）
图存储	Neo4J FlockDB InfoGrid	图形关系的最佳存储。使用传统关系数据库来解决的话性能低下，而且设计使用不方便。
对象存储	db4o Versant	通过类似面向对象语言的语法操作数据库，通过对象的方式存取数据。
xml数据库	Berkeley DB XML BaseX	高效的存储XML数据，并支持XML的内部查询语法，比如XQuery,Xpath。

资源管理层

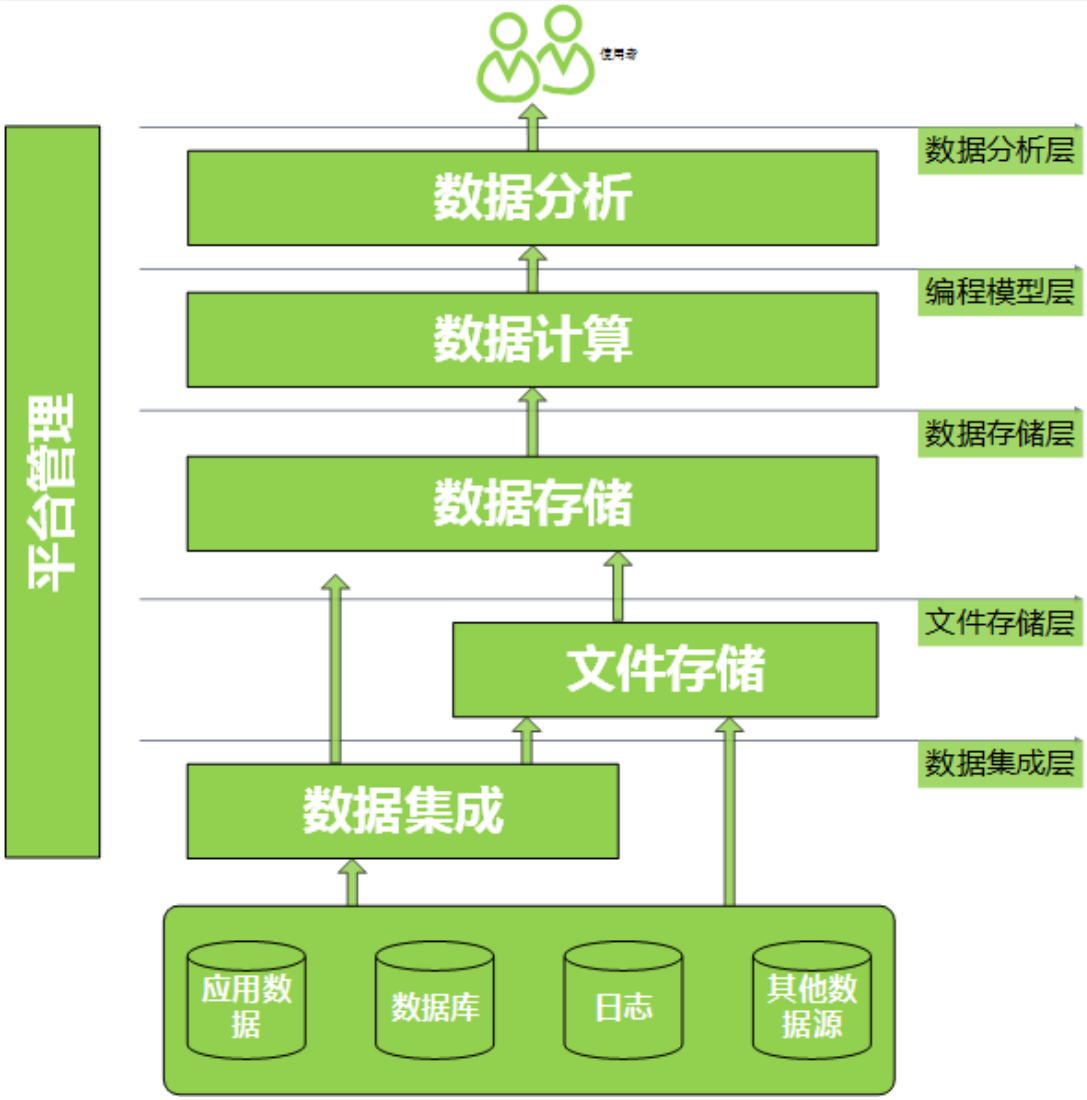
- **资源管理器 (Cluster Manager)**
  Yarn & **Mesos** (UCB学术论文NSDI) **& Borg** (Google)

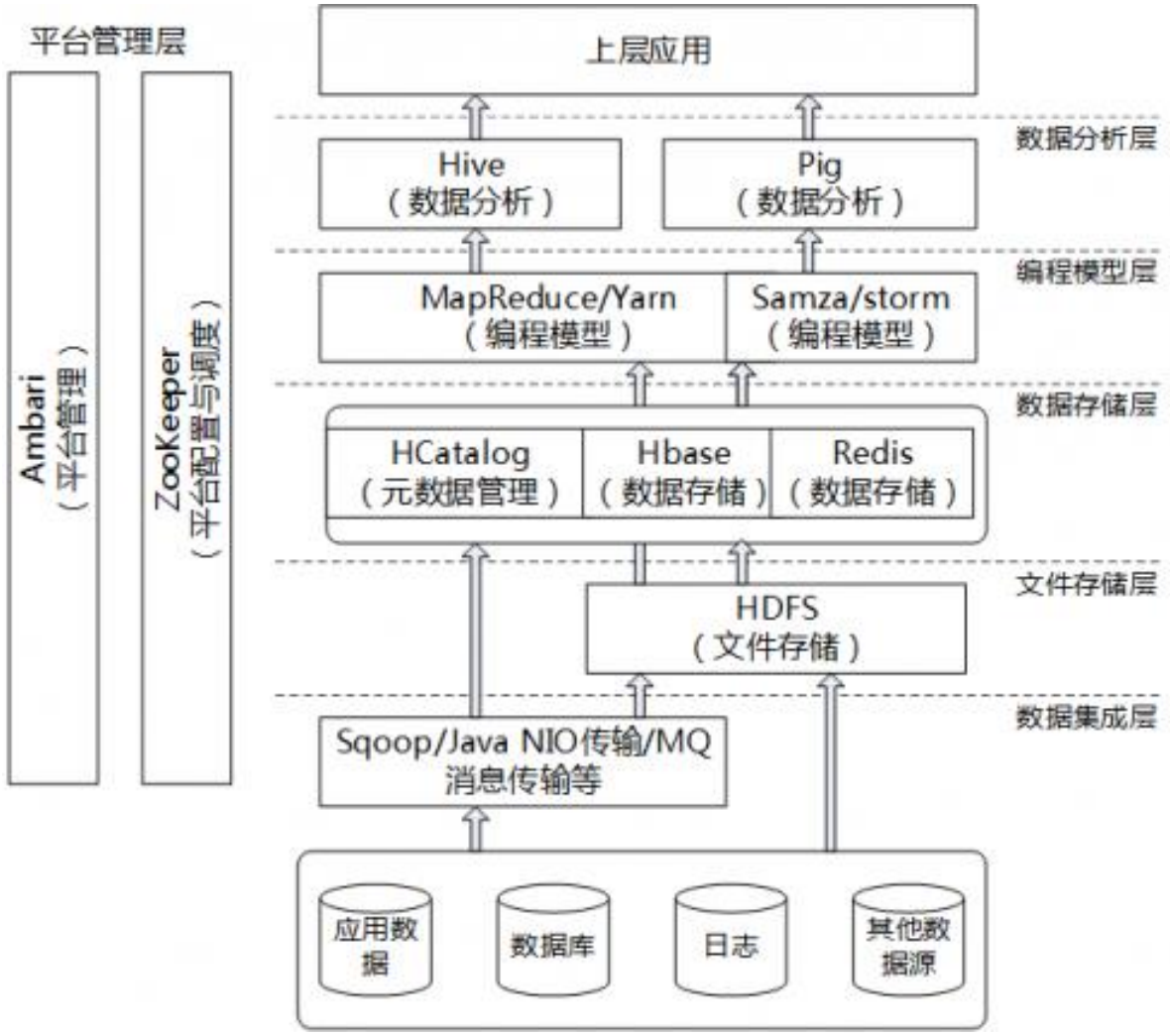
- **调度器 (Scheduler)**
  Capacity Scheduler
  FairShare Scheduler
  Delayed Scheduler
  (Infocom 12-14年，从理论模型到实现，众多论文)

- **协调器 (Coordinator)**
  Zab: 按发射顺序一致 模型 (Apache ZooKeeper)
  Raft: 与Zab类似模型
  Paxos: 要求执行顺序一致模型 (Google Chubby)
  Lamport (The Part-time Parliament 1990)
  (Paxos Made simple ACM TOCS 1998)

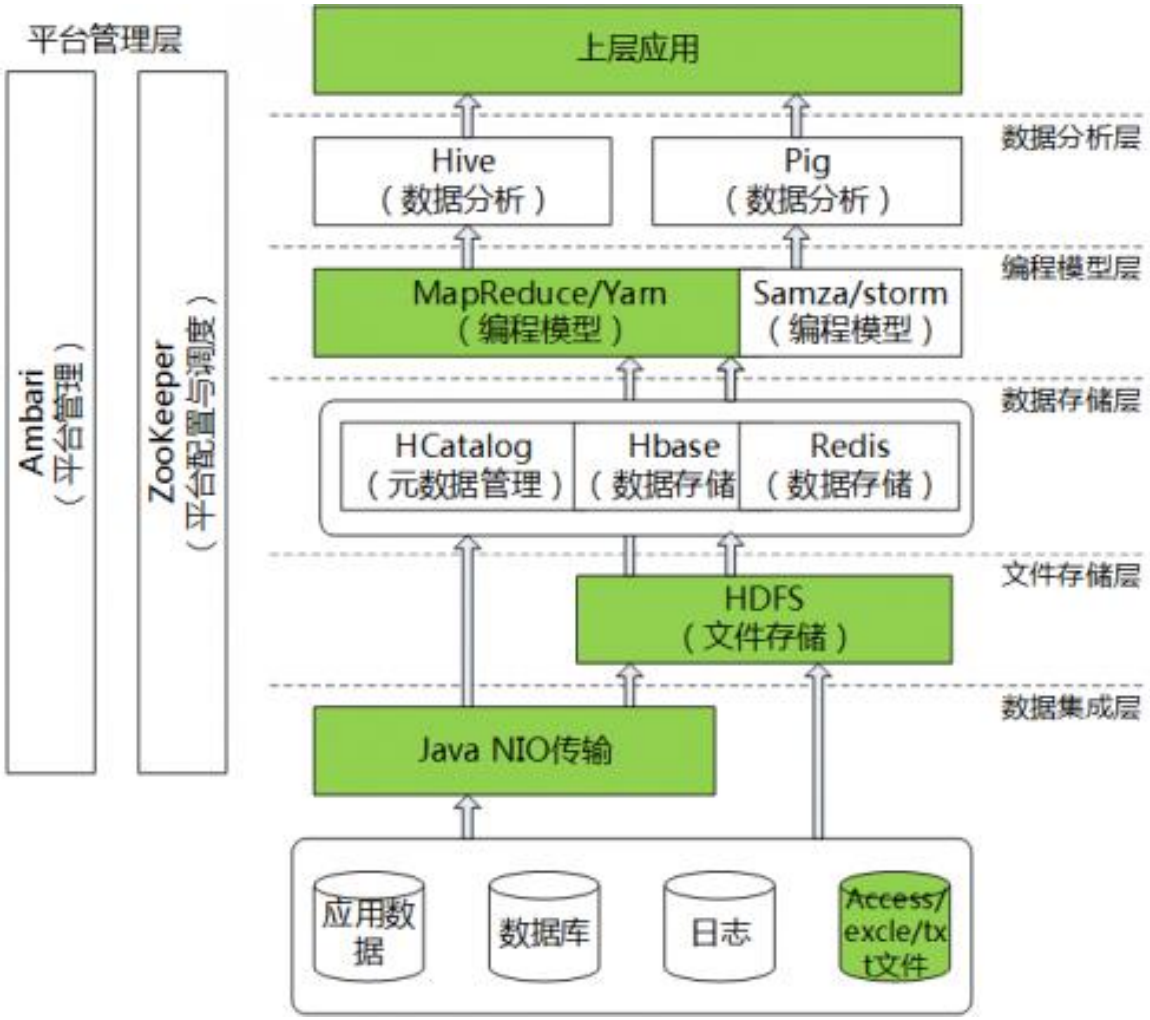
计算框架

分类	框架	描述	文献
运行时	Spark	内存计算框架	UC伯克利
	Flink	类似于Spark	Apache
批处理	MapReduce		
迭代式	Pregel	大规模图处理	Google
	Giraph	Pregel的开源版	
	GraphX	图并行+数据并行	UC伯克利
	Hama	基于BSP的分布式计算引擎	
流式	Storm	大数据实时处理	Twitter
	Samza	流式处理框架	LinkedIn
	Spark流	数据片的批处理	SOSP'13
交互式	Dremel	大数据交互式处理	Google
	Drill	Dremel的开源版	
	Shark	Hive on Spark ($HQL \rightarrow RDD$)	SIGMOD'12
	Dryad	基于DAG的并行数据流水线	EuroSys'07
	Tez	利用Yarn对Dryad的开源实现	Hortonworks
实时	Druid	开源分布式实时分析存储系统	SIGMOD'14
	Piont	开源分布式OLAP系统	LinkedIn

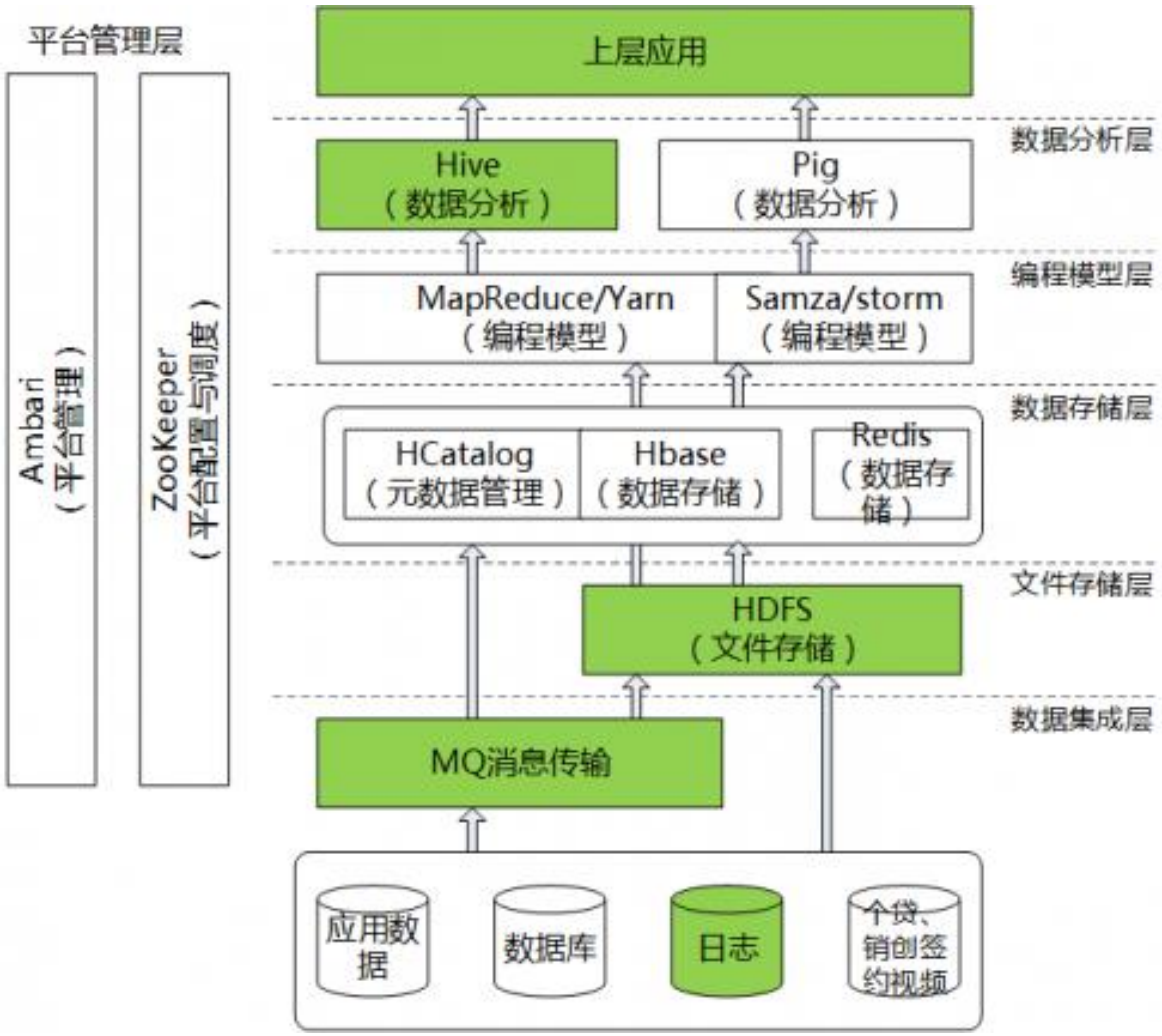

平台管理
数据分析层
数据分析
编程模型层
数据计算
数据存储层
数据存储
文件存储层
文件存储
数据集成层
数据集成
应用数据
数据库
日志
其他数据源


平台管理层
Ambari
（平台管理）
ZooKeeper
（平台配置与调度）
上层应用
数据分析层
Hive
（数据分析）
Pig
（数据分析）
编程模型层
MapReduce/Yarn
（编程模型）
Samza/storm
（编程模型）
数据存储层
HCatalog
（元数据管理）
Hbase
（数据存储）
Redis
（数据存储）
文件存储层
HDFS
（文件存储）
数据集成层
Sqoop/Java NIO传输/MQ
消息传输等
应用数据
数据库
日志
其他数据源

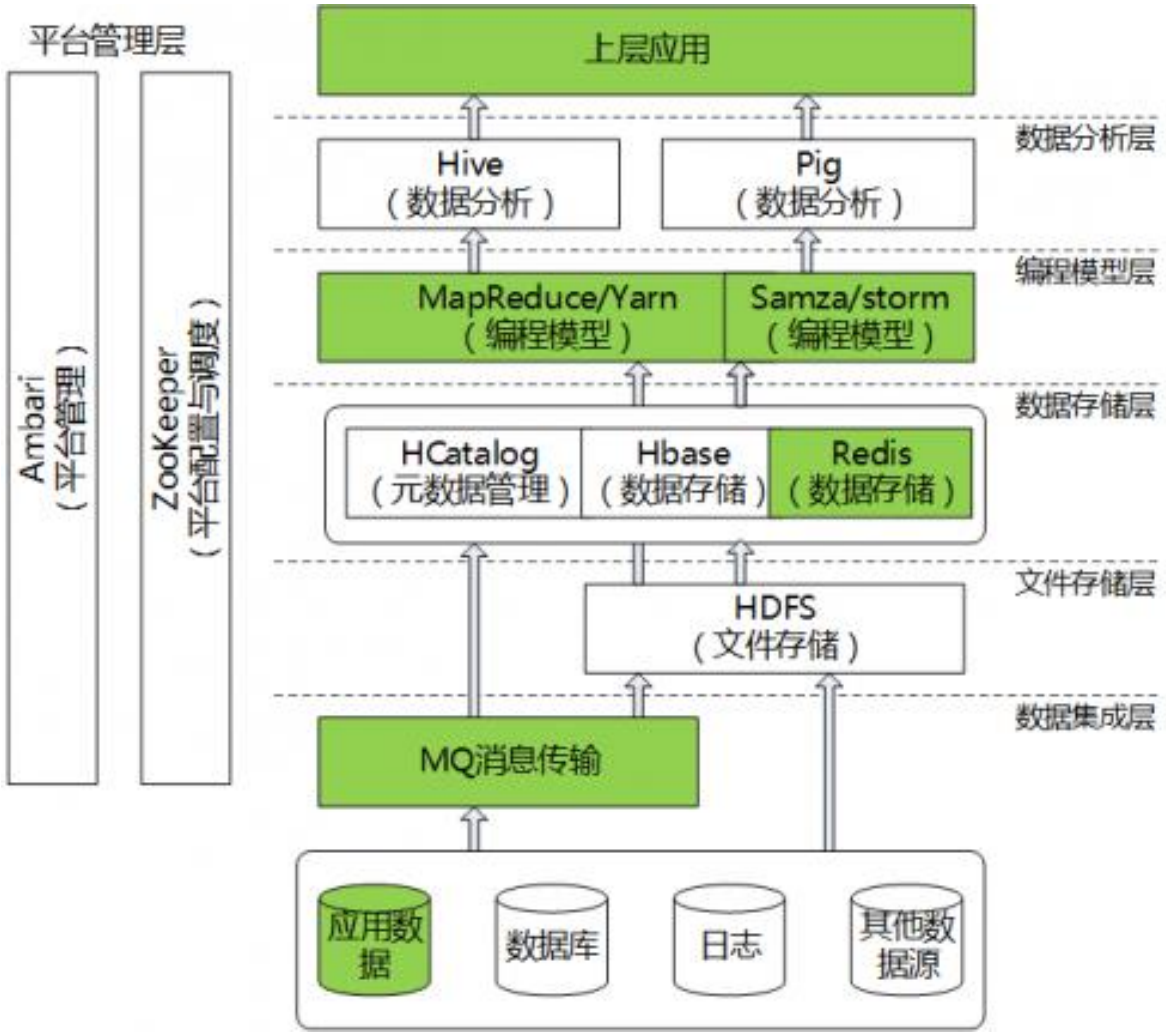
数据分析


平台管理层
Ambari（平台管理）
ZooKeeper（平台配置与调度）
上层应用
Hive（数据分析）
Pig（数据分析）
数据分析层
MapReduce/Yarn（编程模型）
Samza/storm（编程模型）
编程模型层
HCatalog（元数据管理）
Hbase（数据存储
Redis（数据存储）
数据存储层
HDFS（文件存储）
文件存储层
Java NIO传输
数据集成层
应用数据
数据库
日志
Access/excle/txt文件

日志分析


平台管理层
Ambari
（平台管理）
ZooKeeper
（平台配置与调度）
上层应用
数据分析层
Hive
（数据分析）
Pig
（数据分析）
编程模型层
MapReduce/Yarn
（编程模型）
Samza/storm
（编程模型）
数据存储层
HCatalog
（元数据管理）
Hbase
（数据存储）
Redis
（数据存储）
文件存储层
HDFS
（文件存储）
数据集成层
MQ消息传输
应用数据
数据库
日志
个贷、销创签约视频

在线分析


平台管理层
Ambari（平台管理）
ZooKeeper（平台配置与调度）
上层应用
数据分析层
Hive（数据分析）
Pig（数据分析）
编程模型层
MapReduce/Yarn（编程模型）
Samza/storm（编程模型）
数据存储层
HCatalog（元数据管理）
Hbase（数据存储）
Redis（数据存储）
文件存储层
HDFS（文件存储）
数据集成层
MQ消息传输
应用数据
数据库
日志
其他数据源

OLAP架构需求

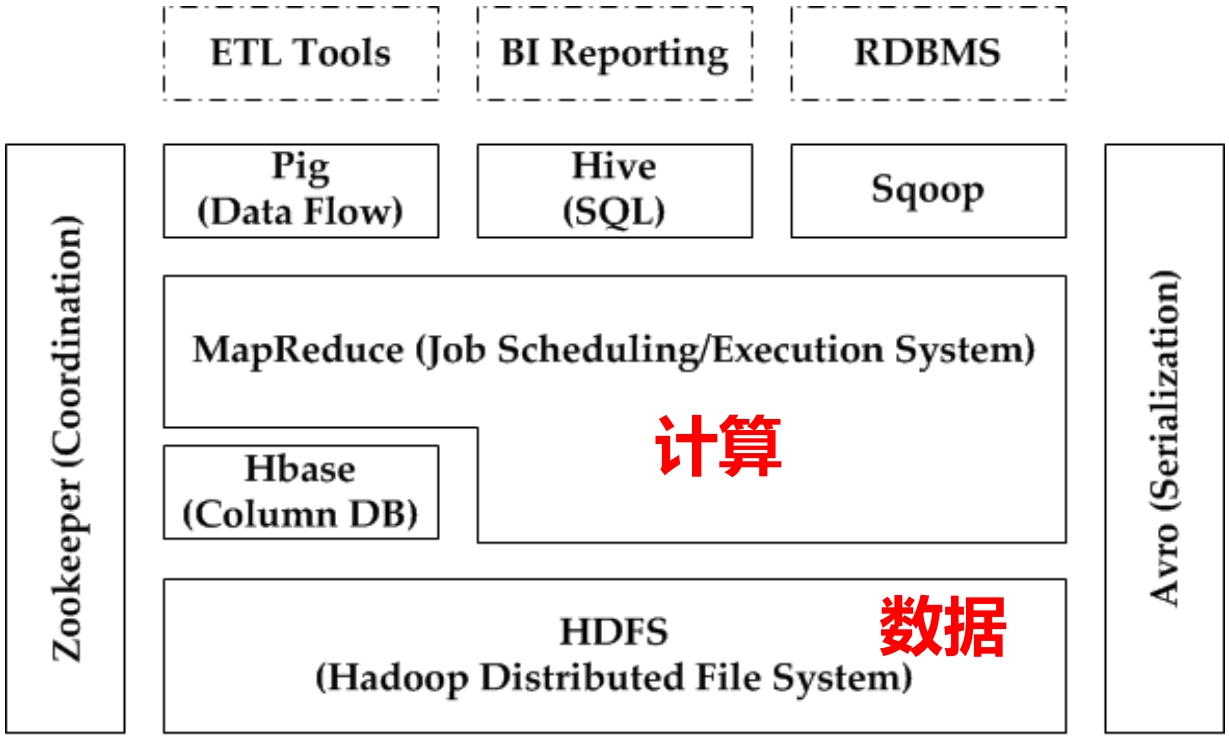
- 可扩展性
- 高可用性
- 高吞吐量计算
- 高效廉价
- 数据备份及鲁棒性
- 应用匹配
  - 读写特性、实时性要求、删除修改等操作
  - I/O密集型、CPU密集型、高并发等

Google架构

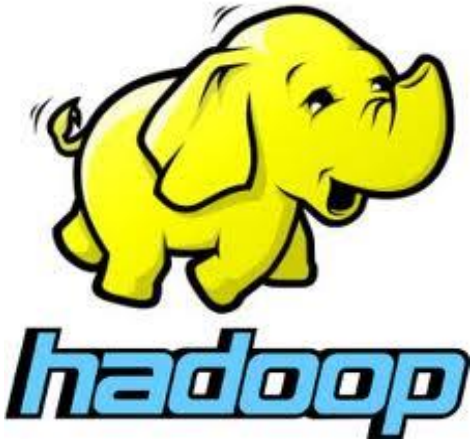
- **Google：架构提出者**
  - Google文件系统GFS(Google File System)
  - 并行数据处理 MapReduce
  - 结构化数据表 BigTable
  - 分布式锁管理 Chubby

处理架构

- Yahoo, Facebook, 阿里云,百度：Hadoop架构；



The Hadoop Ecosystem

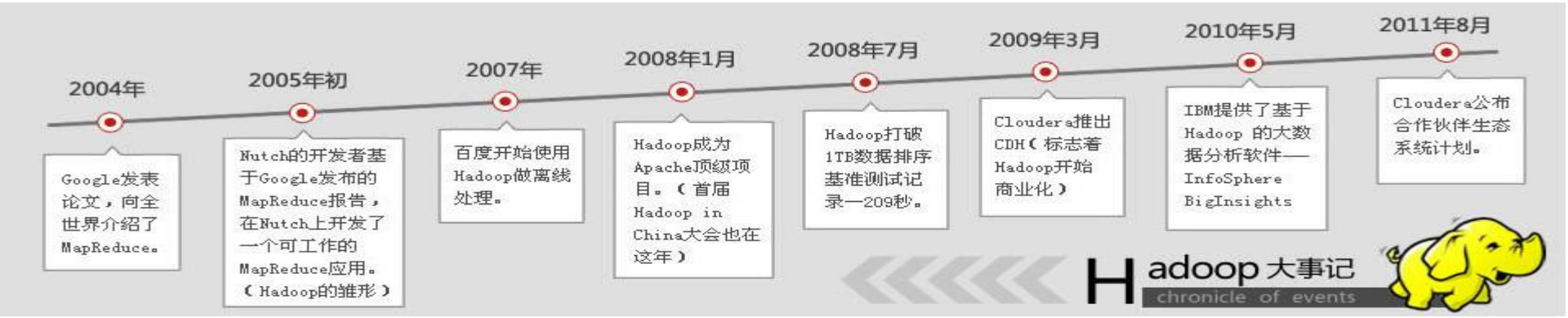


Hadoop

- Yahoo!是最主要的贡献者
- 一个分布式文件系统和并行执行环境
- 让用户便捷地处理海量数据
- Apache软件基金会下面的一个开源项目

Hadoop大事记

- 2004年-- 最初的版本(现在称为HDFS和MapReduce)由Doug Cutting和Mike Cafarella开始实施。
- 2006年2月-- 雅虎的网格计算团队采用Hadoop。
- 08年4月-- 赢得世界最快1 TB数据排序在900个节点上用时209秒。
- 09年3月-- 17个集群总共24 000台机器。
- 12年4月-- Hadoop 稳定版1.0.2。
- 17年1月– Hadoop 3.0.0 Alpha2



Hadoop的特点

- **扩容能力**（Scalable）：能可靠地（reliably）存储和处理千兆字节（PB）数据
- **成本低**（Economical）：可以通过普通机器组成的服务器群来分发以及处理数据。这些服务器群总计可达数千个节点
- **高效率**（Efficient）：通过分发数据，Hadoop可以在数据所在的节点上并行地处理它们，这使得处理非常的快速
- **可靠性**（Reliable）：Hadoop能自动地维护数据的多份复制，并且在任务失败后能自动地重新部署（redeploy）计算任务。

选择合适的工具

Use The Right Tool For The Right Job

Hadoop:

When to use?

- Affordable Storage/ Compute
- Structured or Not (Agility)
- Resilient Auto Scalability

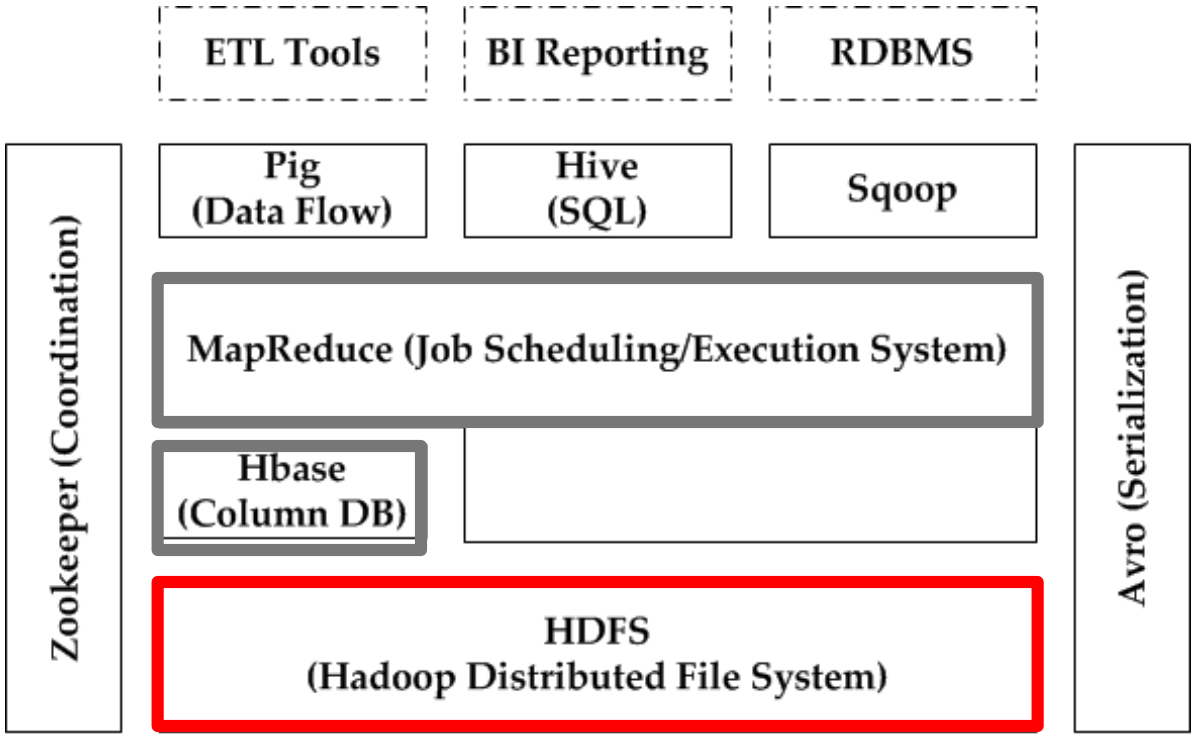
Relational Databases:

When to use?

- Interactive Reporting (<1sec)
- Multistep Transactions
- Interoperability

技术内容

- HDFS — 块存储，分布式架构
- HBase — 列存储，Key-Value数据库
- MapReduce — 计算模型



The Hadoop Ecosystem

设计GFS的动机

- Google需要一个支持海量存储的文件系统
  - 购置昂贵的分布式文件系统与硬件?

是否可以在一堆廉价且不可靠的硬件上构建可靠的分布式文件系统?

设计GFS的动机

- 为什么不使用当时现存的文件系统？Google所面临的问题与众不同
  - 组件失败成为一种常态而不是异常；
  - 与传统的标准相比，文件是巨大的；
  - 大部分的文件更新模式是通过在尾部追加数据而不是覆盖现有数据。文件内部的随机写操作几乎是不存在的。一旦写完，文件就是只读的，而且通常是顺序读；
  - 应用程序和文件系统API的协同设计，增加了整个系统的灵活性。比如通过放松了GFS的一致性模型大大简化了文件系统，同时也没有给应用程序带来繁重的负担；
  - 高的持续带宽比低延时更重要。

- **GFS的设计思路**

  – 购置昂贵的分布式文件系统与硬件?

  – 还是可以在一堆廉价且不可靠的硬件上构建可靠的分布式文件系统?

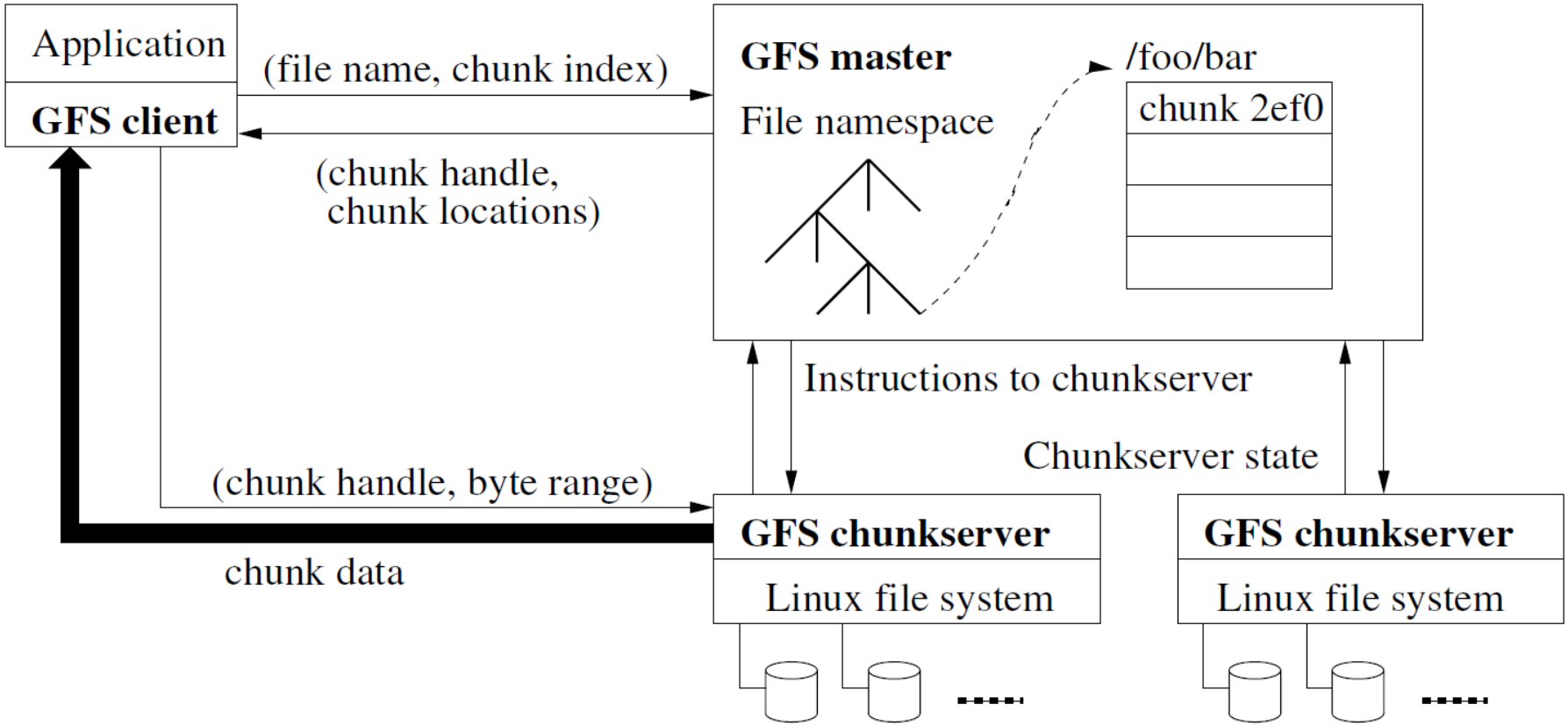
- **GFS的假设与目标**
  - 硬件出错是正常而非异常
    - 系统应当由大量廉价、易损的硬件组成
    - 必须保持文件系统整体的可靠性
  - 主要负载是流数据读写
    - 主要用于程序处理批量数据，而非与用户的交互或随机 读写
    - 数据写主要是“追加写”，“插入写”非常少
  - 需要存储大尺寸的文件
    - 存储的文件尺寸可能是GB或TB量级，而且应当能支持存 储成千上万的大尺寸文件

- **GFS的实现技巧**
  - 将文件划分为若干块（Chunk）存储
    - 每个块固定大小（64M）
  - 通过冗余来提高可靠性
    - 每个数据块至少在3个数据块服务器上冗余
    - 数据块损坏概率?
  - 通过单个master来协调数据访问、元数据存储
    - 结构简单，容易保持元数据一致性
  - 无缓存
    - Why?

- **文件分块存储**

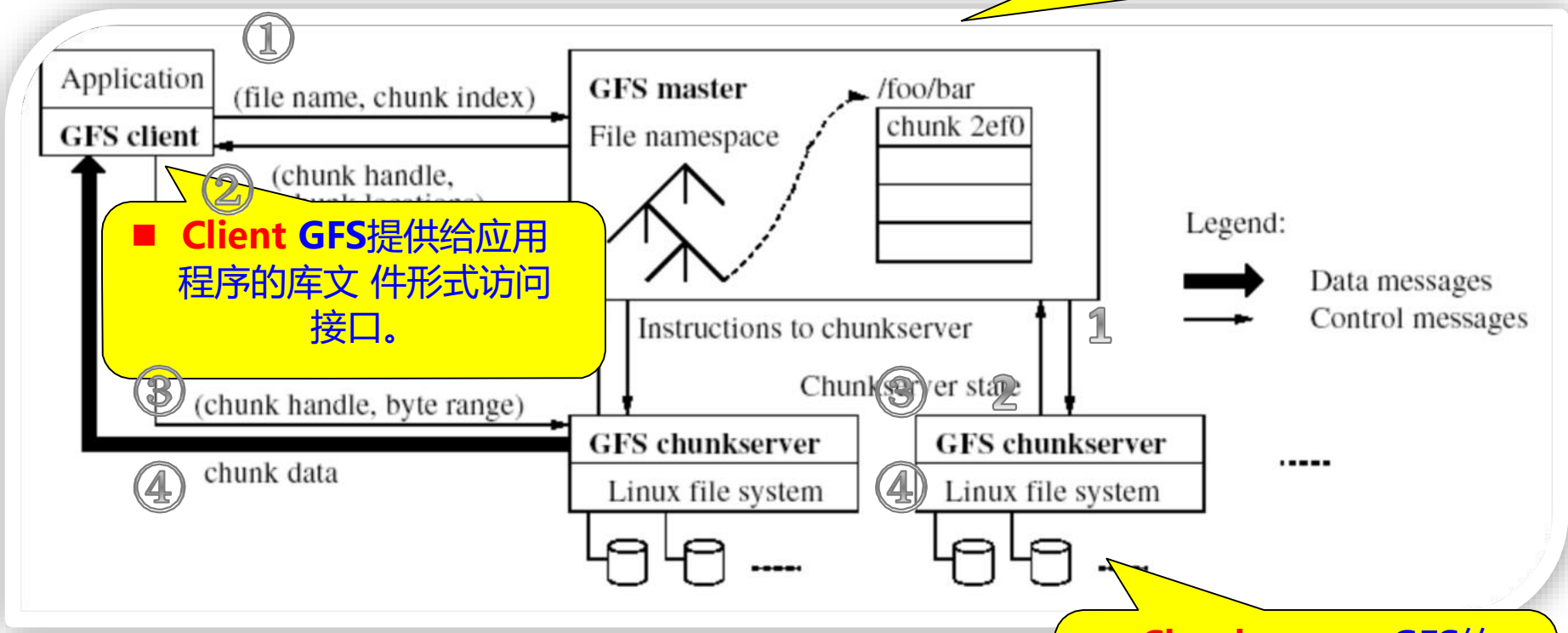
A	B	C	(free space)		
A	B	C	A	(free space)	
A	(free space)	C	A	(free space)	
A	D	C	A	D	(free)

GFS结构


Application
GFS client
(file name, chunk index)
(chunk handle,
chunk locations)
GFS master
File namespace
/foo/bar
chunk 2ef0
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
GFS chunkserver
Linux file system
GFS chunkserver
Linux file system

• GFS架构

■ **Master GFS**的管理节点，逻辑上只 有一个，保存系统元数据。



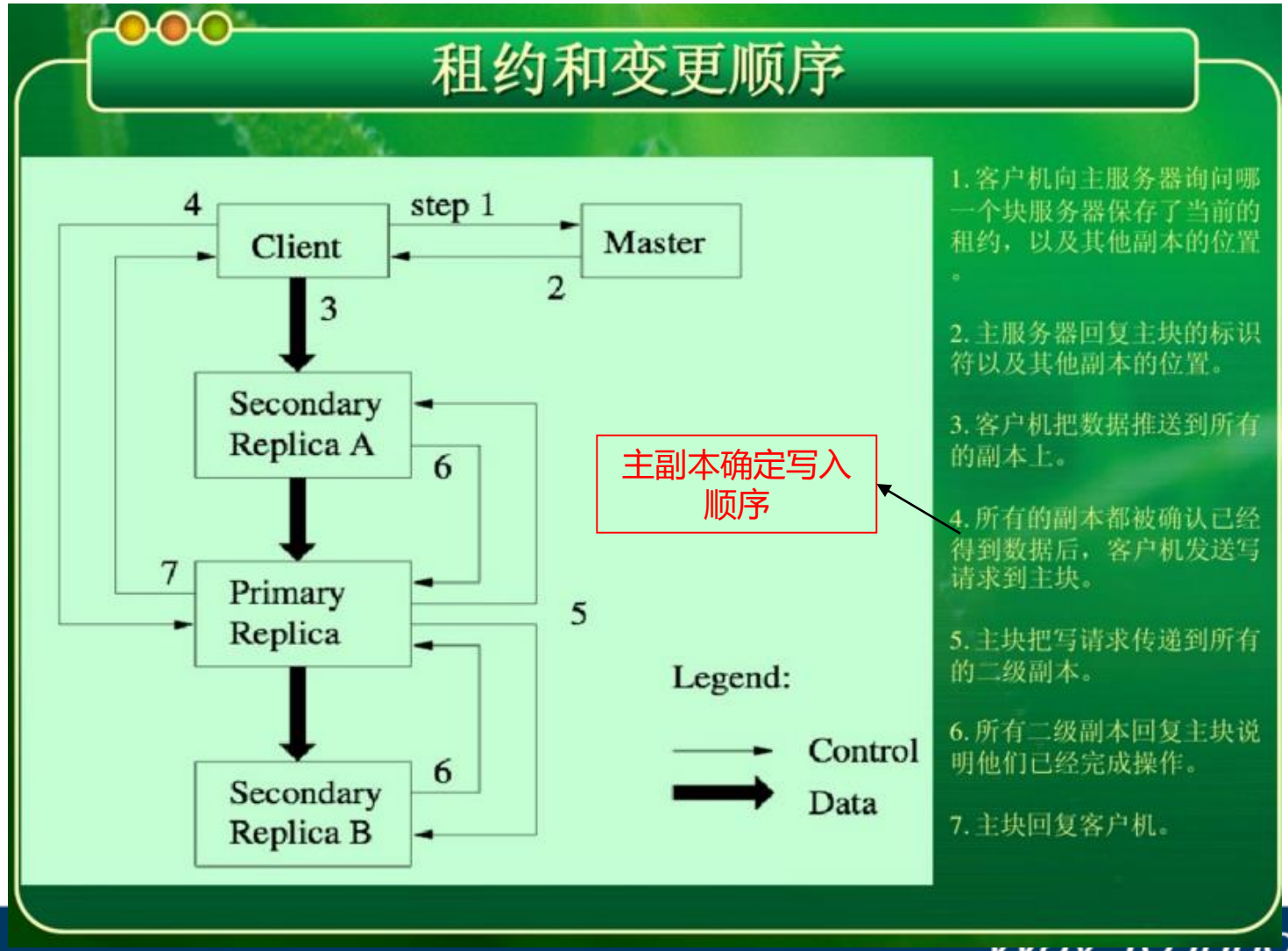
■ **Client GFS**提供给应用程序的库文 件形式访问接口。

■ **Chunk server GFS**的数据存储节点，大小 固定（**64MB**）。

GFS数据一致性

- 同一时刻多个客户端并发写入或追加同一文件出现数据不一致性问题
- GFS 使用租约机制来保障在跨多个副本的数据写入中保持顺序一致性。
  - GFS Master 将 chunk 租约发放给其中一个副本，这个副本我们就称为主副本，其他副本称为次副本。
  - 主副本确定一个针对该 chunk 的写入顺序，次副本则遵守这个顺序，这样就保障了全局顺序一致性。
  - 租约机制的设计主要是为了减轻 Master 的负担，由主副本所在的 chunk server 来承担流水线顺序的安排。

- GFS写流程


租约和变更顺序
4
step 1
Client
Master
2
3
Secondary
Replica A
6
7
Primary
Replica
5
6
Secondary
Replica B
Legend:
Control
Data
主副本确定写入
顺序
1. 客户机向主服务器询问哪一个块服务器保存了当前的租约，以及其他副本的位置。
2. 主服务器回复主块的标识符以及其他副本的位置。
3. 客户机把数据推送到所有的副本上。
4. 所有的副本都被确认已经得到数据后，客户机发送写请求到主块。
5. 主块把写请求传递到所有的二级副本。
6. 所有二级副本回复主块说明他们已经完成操作。
7. 主块回复客户机。

- **GFS系统特点**
  - 采用中心服务器模式
    - 可以方便地增加Chunk Server
    - Master掌握系统内所有Chunk Server的情况，方便进行负载均衡
    - 不存在元数据的一致性问题

- **GFS系统特点**
  - 不缓存数据
    - GFS的文件操作大部分是流式读写，不存在大量的重复读写，使用Cache对性能提高不大
    - Chunk Server上的数据存取使用本地文件系统，如果某个Chunk读取频繁，文件系统具有Cache
    - 从可行性看，Cache与实际数据一致性维护也极其复杂

- **GFS系统特点**

– 在用户态下实现

•直接利用Chunk Server的文件系统存取Chunk，实现简单

• 用户态应用调试较为简单，利于开发

•用户态的GFS不会影响Chunk Server的稳定性

– 提供专用的访问接口

• 未提供标准的POSIX访问接口

• 降低GFS的实现复杂度

- Master节点任务
  - **存储元数据**
  - **文件系统目录管理与加锁**
  - **与ChunkServer进行周期性通信**
    - 发送指令，搜集状态，跟踪数据块的完好性
  - **数据块创建、复制及负载均衡**
    - 对ChunkServer的空间使用和访问速度进行负载均衡， 平滑数据存储和访问请求的负载
    - 对数据块进行复制、分散到ChunkServer上
    - 一旦数据块冗余数小于最低数，就发起复制操作

- **Master节点任务**
  - **垃圾回收**
    - 在日志中记录删除操作，并将文件改名隐藏
    - 缓慢地回收隐藏文件
    - 与传统文件删除相比更简单、更安全
  - **陈旧数据块删除**
    - 探测陈旧的数据块，并删除

- **Master节点瓶颈**
  - 尽可能减少数据存取中Master的参与程度
  - 不使用Master读取数据，仅用于保存元数据
  - 客户端缓存元数据
  - 采用大尺寸的数据块（64M）
  - 数据修改交由Primary Chunk Server完成

- **GFS的容错机制**
  - Master容错
    - 三类元数据：命名空间（目录结构）、Chunk与文件名的映射以及Chunk副本的位置信息
    - 前两类通过日志提供容错，Chunk副本信息存储于ChunkServer，Master出现故障时可恢复

- **GFS的容错机制**
  - Chunk Server容错
    - 每个Chunk有多个存储副本（通常是3个），分别存储于不同的服务器上
    - 每个Chunk又划分为若干Block（64KB），每个 Block对应一个32bit的校验码，保证数据正确（若某个Block错误，则转移至其他Chunk副本）

- **GFS的快照机制**

快照（Snapshot）操作是对源文件/目录进行一个“快照”操作，生成该时刻源文件/目录的一个瞬间状态存放与目标文件/目录中。

- GFS中使用标准的copy-on-write机制生成快照
- 为了对某个文件做Snapshot，首先需要停止这个文件的写服务，接着增加这个文件的所有chunk的引用计数，以后修改这些chunk时会拷贝生成新的chunk。
- 对某个文件执行Snapshot的大致步骤如下：
  ① 通过Lease机制收回对文件每一个chunk写权限，停止对文件的写服务；
  ② Master拷贝文件名等元数据生成一个新的Snapshot文件；
  ③ 对执行Snapshot的文件的所有chunk增加引用计数；

- 讨论

– 只有一个主服务器(GFS Master)

– 限制系统可扩展性和可靠性的一个缺陷

– 系统最大存储容量和正常工作时间受制于主服务器的容量和正常工作时间

– 要将所有的元数据进行编制，并且几乎所有的动作和请求都经过它

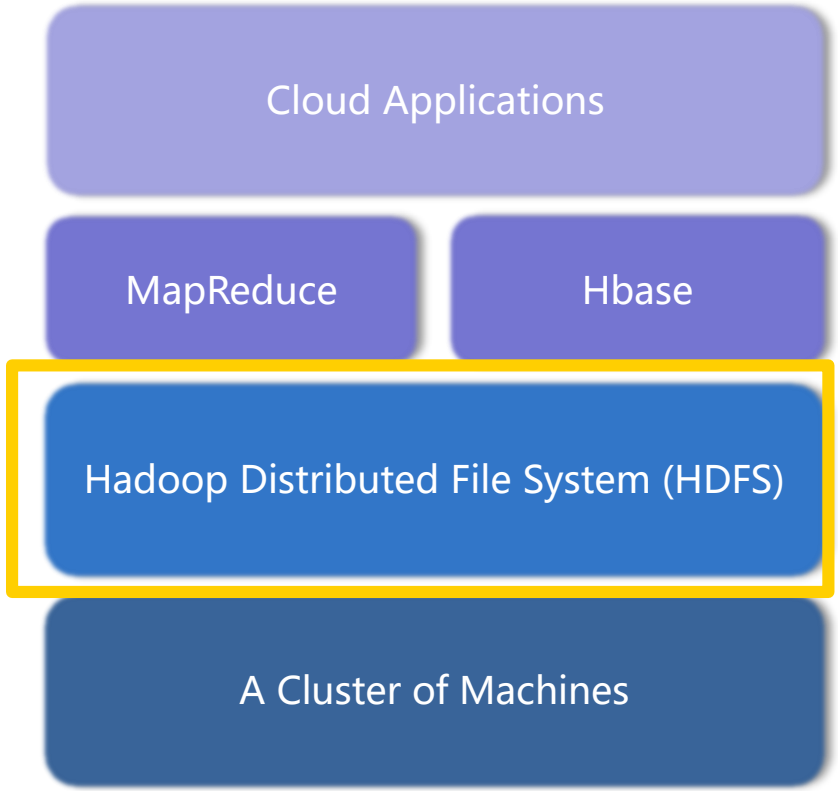
- 讨论
  - 主服务器通常是网络上性能最好的节点之一
  - 可靠性通过“影子”主服务器制作主服务器的镜像，一旦主服务器失败它将接替工作
  - 主服务器极少成为瓶颈，因为客户端仅仅取得元数据然后将它们缓存起来，随后交互直接与chunk servers进行
  - 使用单个主服务器可以大幅度地降低软件复杂性，保证数据完整性、动态操作、负载均衡和安全性。

- GFS在Google中的部署
  - 超过50个GFS集群
    - 每个集群包含数千个存储节点
    - 管理着PB(10^{15}Byte)级的数据

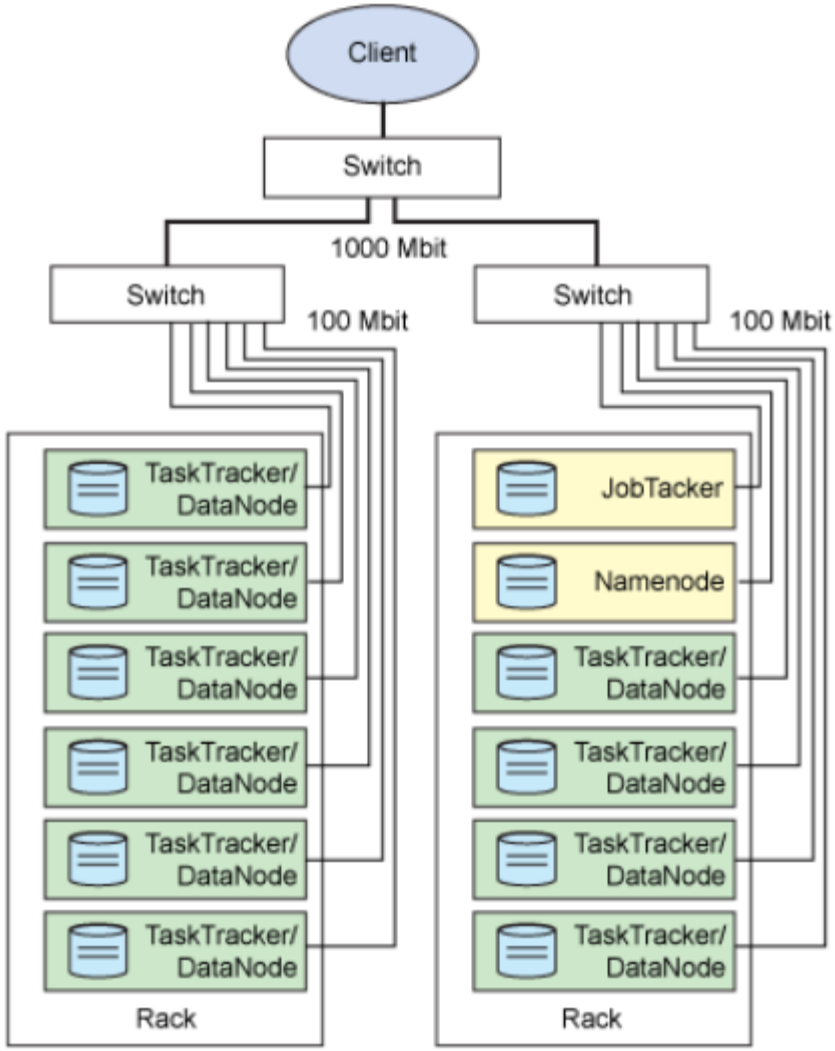
HDFS

概述

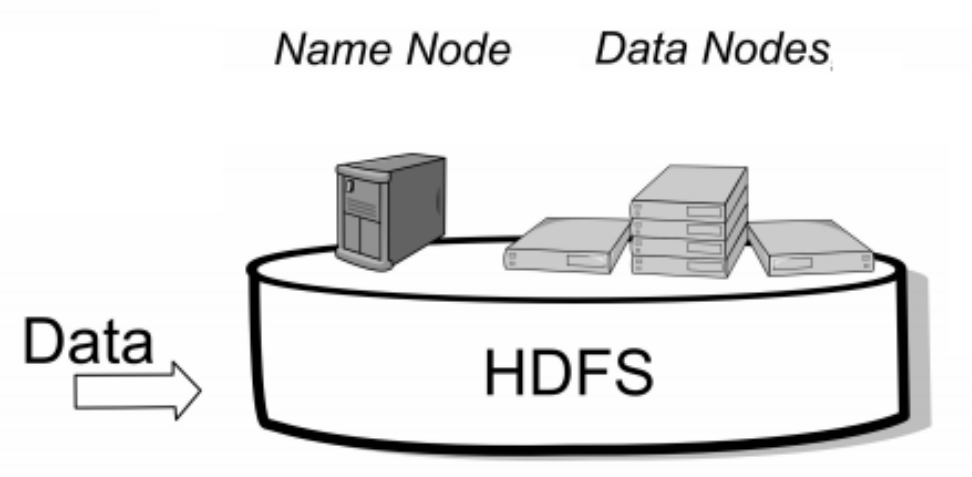
- HDFS(Hadoop Distributed File System).
- 参考GFS设计.
- 面向大数据分析的可扩展分布式文件系统.
- 基于商业硬件.
- 是Hadoop应用主要存储方案.



Hadoop集群


Client
Switch
1000 Mbit
Switch
100 Mbit
Switch
100 Mbit
TaskTracker/
DataNode
TaskTracker/
DataNode
TaskTracker/
DataNode
TaskTracker/
DataNode
TaskTracker/
DataNode
TaskTracker/
DataNode
Rack
JobTacker
Namenode
TaskTracker/
DataNode
TaskTracker/
DataNode
TaskTracker/
DataNode
TaskTracker/
DataNode
Rack

HDFS



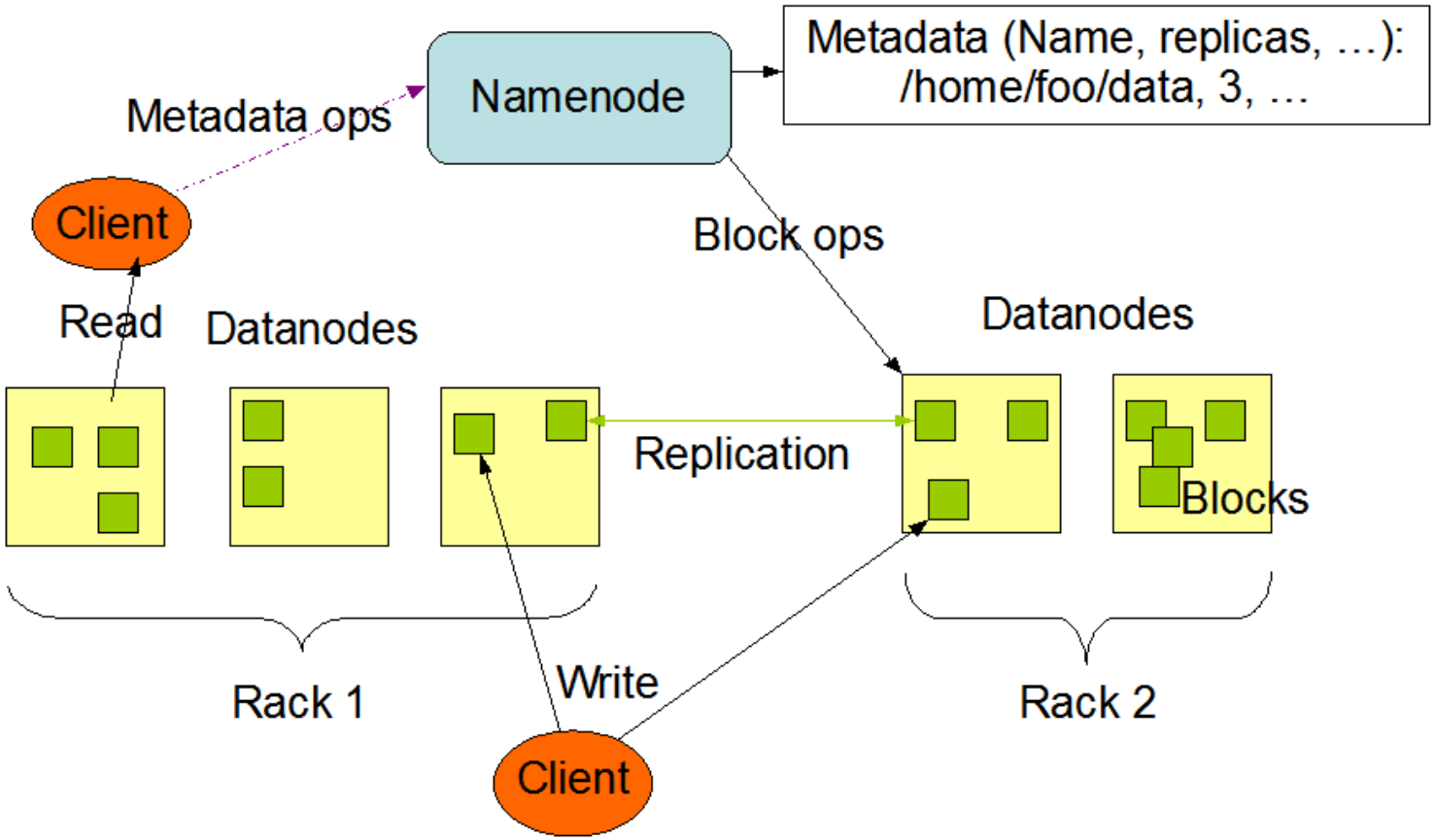
Hadoop Distributed File System (HDFS)

NameNode	DataNode
• 存储元数据	• 存储文件内容
• 元数据保存在内存中	• 文件内容保存在磁盘
• 保存文件,Block ,DataNode之间的映射关系	• 维护了Block ID到DataNode本地文件的映射关系

系统架构

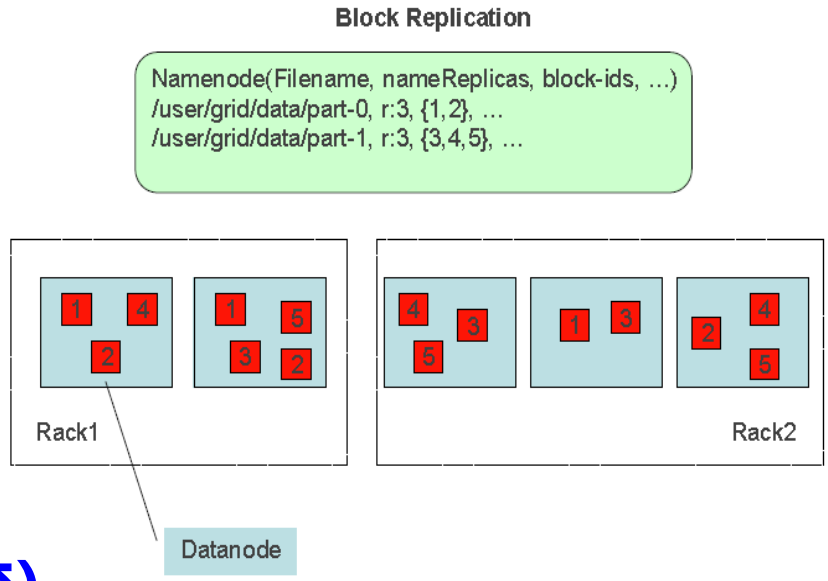
HDFS Architecture

文件和Block名字空间，文件到Block的映射信息，每个Block的备份的位置

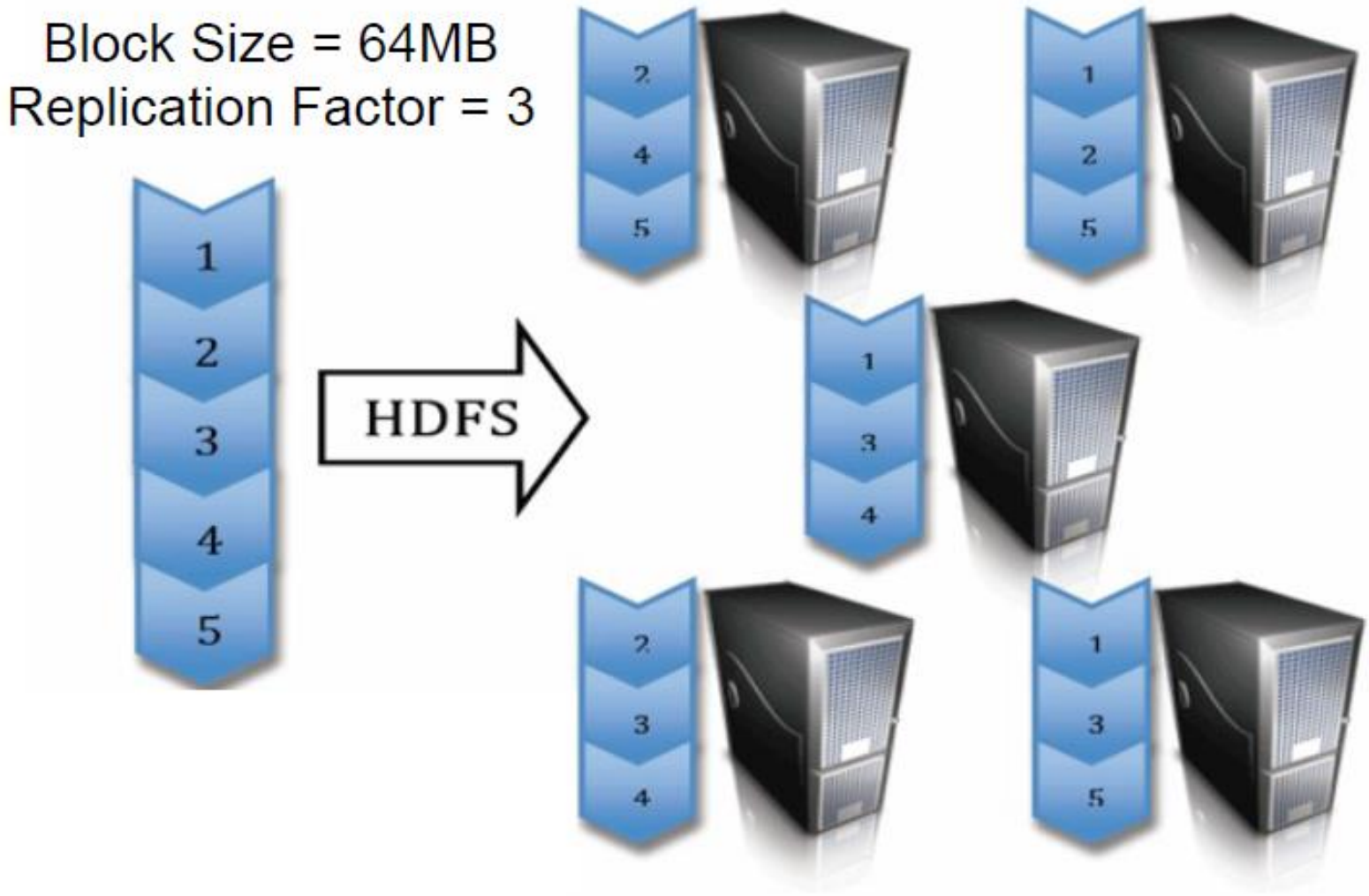


保障可靠性的措施

- 一个名字节点和多个数据节点
- 数据复制（冗余机制）
  --存放的位置（机架感知策略）
- 故障检测
  --数据节点
    心跳包（检测是否宕机）
    块报告（安全模式下检测）
    数据完整性检测（校验和比较）
  --名字节点（日志文件，镜像文件）
- 垃圾回收机制



冗余机制


Block Size = 64MB
Replication Factor = 3
1
2
3
4
5
HDFS
2
4
5
1
2
5
1
3
4
2
3
4
1
3
5

NameNode挂了怎么办?

- **持久化元数据**
  - 元数据主要来源是主节点硬盘上的元数据文件，其中主要包括元数据镜像fsimage和元数据操作日志文件edits。当主节点启动后，所有的元数据被加载到内存中进行管理。
  - HDFS正常运行时，如果文件有所变动，更新将会被写入内存中的元数据结构，并将元数据操作日志记录写入磁盘的日志文件edit log中。
  - 如果主节点关闭（正常关闭、异常关闭）后，内存中的元数据将不复存在，内存的所有元数据信息将会丢失。因此，需要将更新后的元数据文件写到磁盘中永久储存。当启动HDFS时，主节点会将fsimage和edits文件内容读入内存，并进行合并。同时，将生成新的fsimage和edits文件写入磁盘。

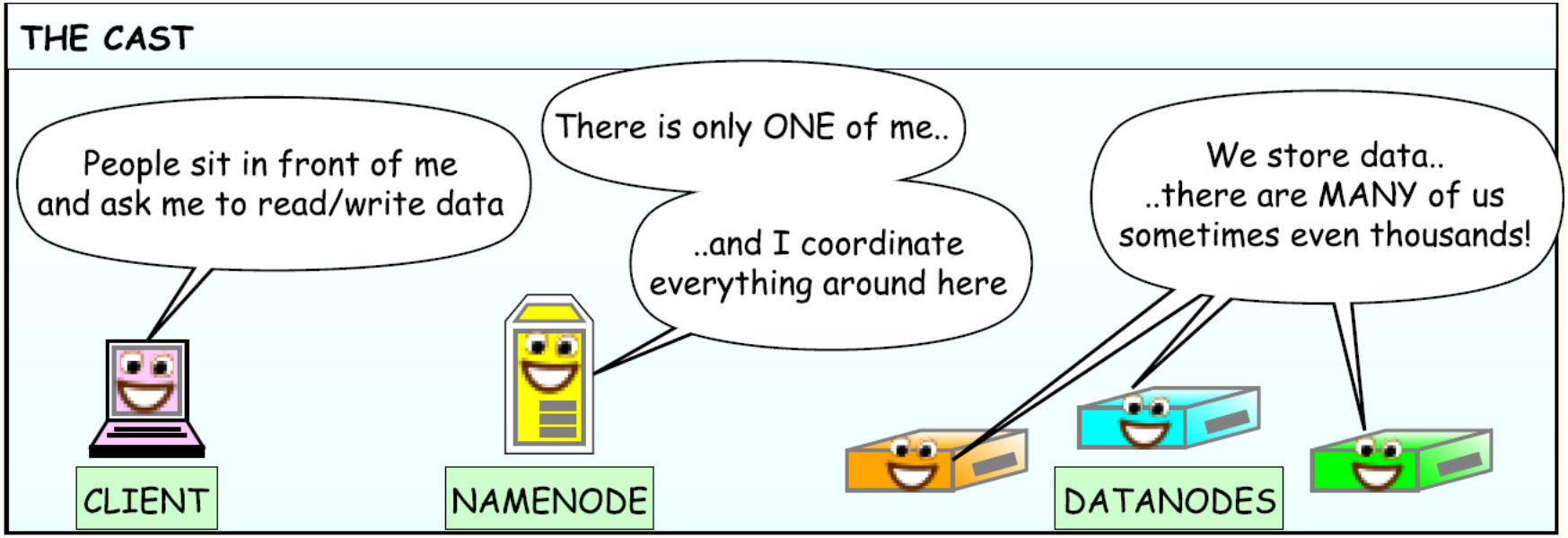
NameNode挂了怎么办?

- Secondary NameNode
  - 将NameNode的fsimage与edit log从Namenode复制到临时目录
  - 将fsimage同edit log合并，并产生新的fsimage （减少启动时间）
  - 将产生的新的fsimage上传给NameNode
  - 清除NameNode中的edit log

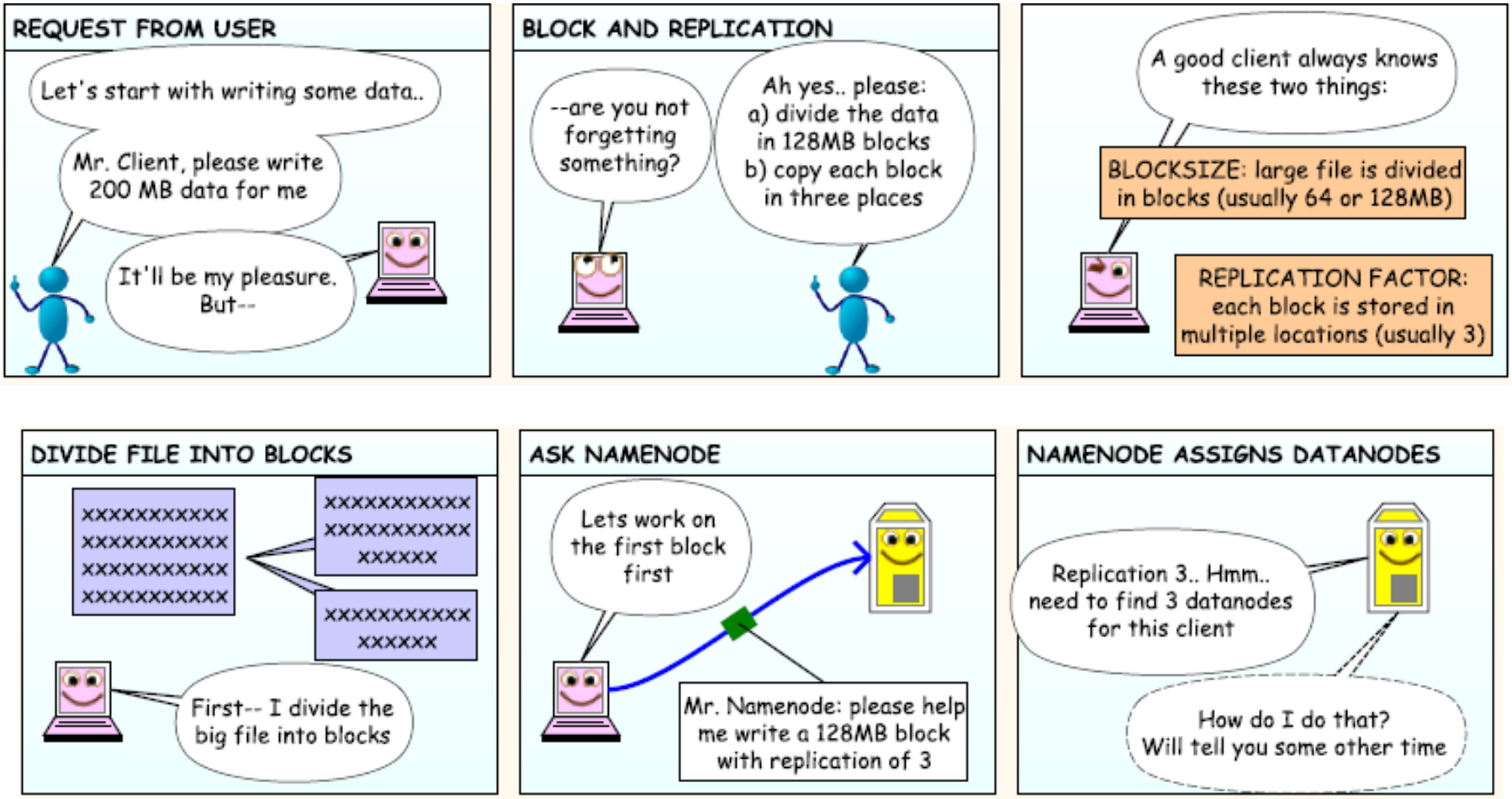
[注]： Secondary NameNode仅仅对NameNode中元数据提供冷备方案

HDFS核心特点

- **大文件和大数据块**：数据块默认大小为64MB，元数据放置在内存中以提升访问效率。
- **操作以添加为主**：因为文件很少被删减或者覆盖，通常只是进行添加或者读取操作，这样能充分考虑到硬盘线性吞吐量大和随机读写慢的特点。
- **支持容错**：采用了单Master的方案，但整个系统会保证每个Master都会有其相对应的复制品，以便于在 Master节点出现问题时进行切换。（节点失效是常态）
- **高吞吐量**：虽然其单个节点的性能无论是从吞吐量还是延迟都很普通，但因为其支持上千的节点，所以数据吞吐量很大。
- **保护数据**：首先，文件被分割成固定尺寸的数据块以便于保存，而且每个数据块都会被系统复制三份。
- **扩展能力强**：因为元数据偏小，使得一个Master节点能控制上千个存储数据的DataNode节点。
- **支持压缩**：对于那些稍旧的文件，可以通过对它进行压缩，来节省硬盘空间，并且压缩率非常惊人，有时甚至接近90%。

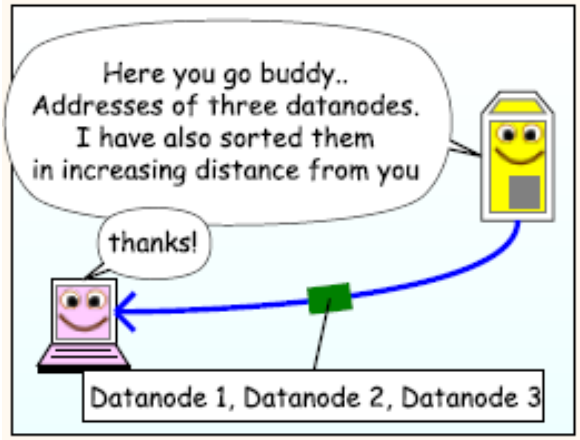

THE CAST
People sit in front of me and ask me to read/write data
There is only ONE of me..
..and I coordinate everything around here
We store data..
..there are MANY of us sometimes even thousands!
CLIENT
NAMENODE
DATANODES

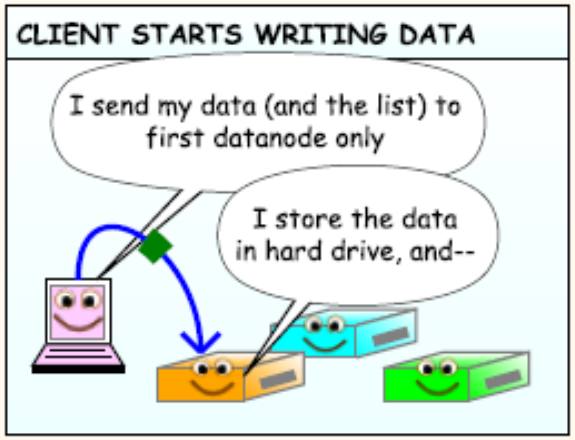
HDFS写过程(1)

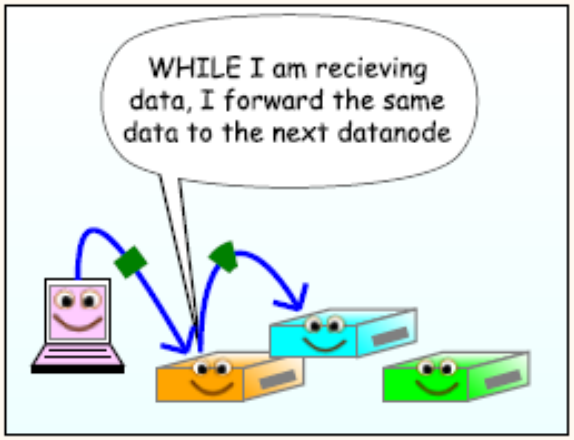

REQUEST FROM USER
Let's start with writing some data..
Mr. Client, please write 200 MB data for me
It'll be my pleasure. But--
BLOCK AND REPLICATION
--are you not forgetting something?
Ah yes.. please: a) divide the data in 128MB blocks b) copy each block in three places
A good client always knows these two things:
BLOCKSIZE: large file is divided in blocks (usually 64 or 128MB)
REPLICATION FACTOR: each block is stored in multiple locations (usually 3)
DIVIDE FILE INTO BLOCKS
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxx
xxxxxxxxxxx
xxxxxx
First-- I divide the big file into blocks
ASK NAMENODE
Lets work on the first block first
Mr. Namenode: please help me write a 128MB block with replication of 3
NAMENODE ASSIGNS DATANODES
Replication 3.. Hmm.. need to find 3 datanodes for this client
How do I do that? Will tell you some other time

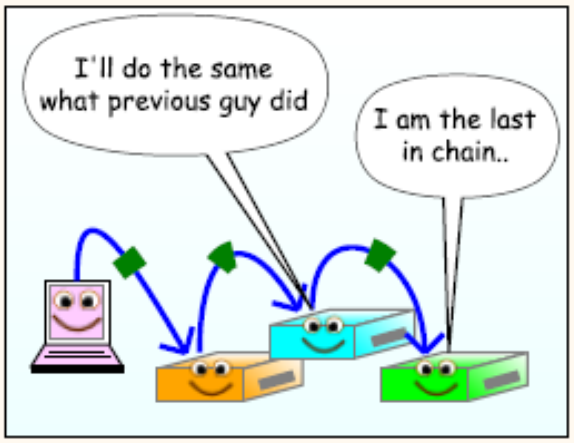
63

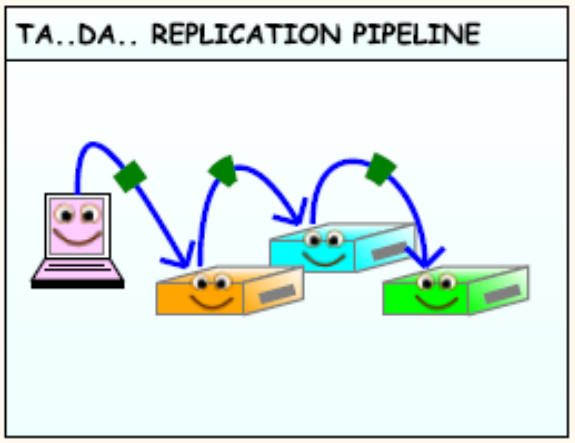
HDFS写过程(2)

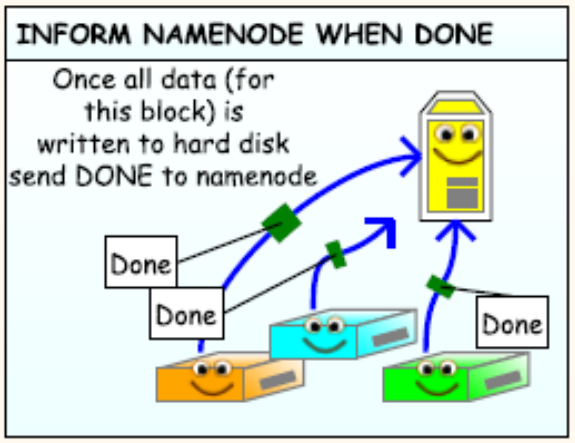

Here you go buddy..
Addresses of three datanodes.
I have also sorted them
in increasing distance from you
thanks!
Datanode 1, Datanode 2, Datanode 3


CLIENT STARTS WRITING DATA
I send my data (and the list) to
first datanode only
I store the data
in hard drive, and--

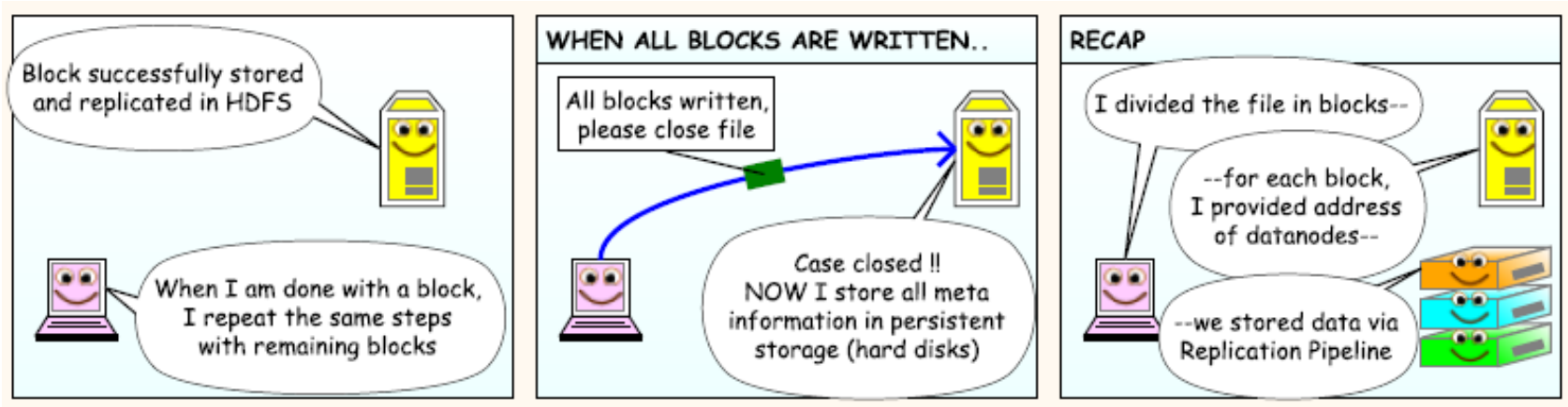

WHILE I am recieving
data, I forward the same
data to the next datanode


I'll do the same
what previous guy did
I am the last
in chain..


TA..DA.. REPLICATION PIPELINE

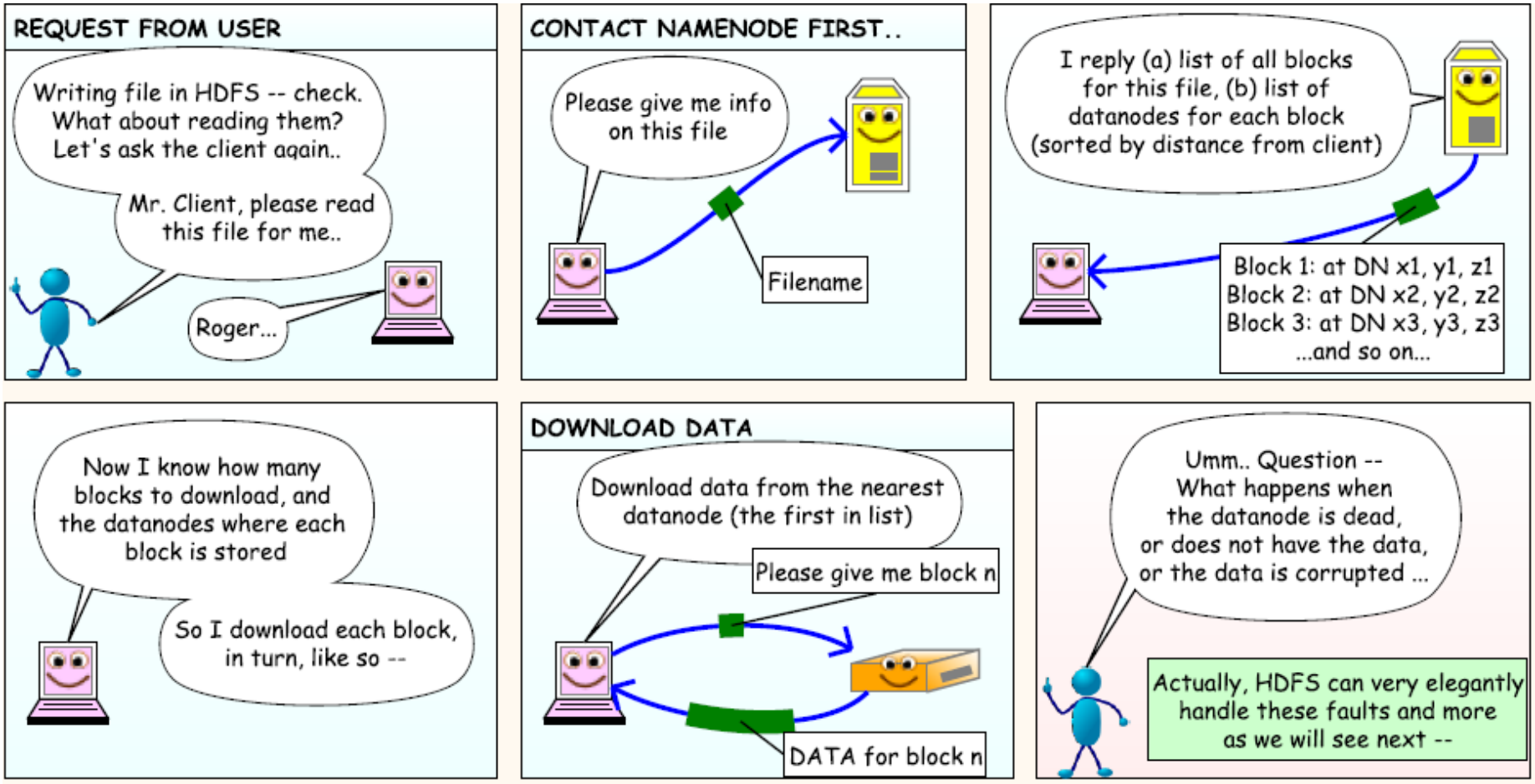

INFORM NAMENODE WHEN DONE
Once all data (for
this block) is
written to hard disk
send DONE to namenode
Done
Done
Done

HDFS写过程(3)

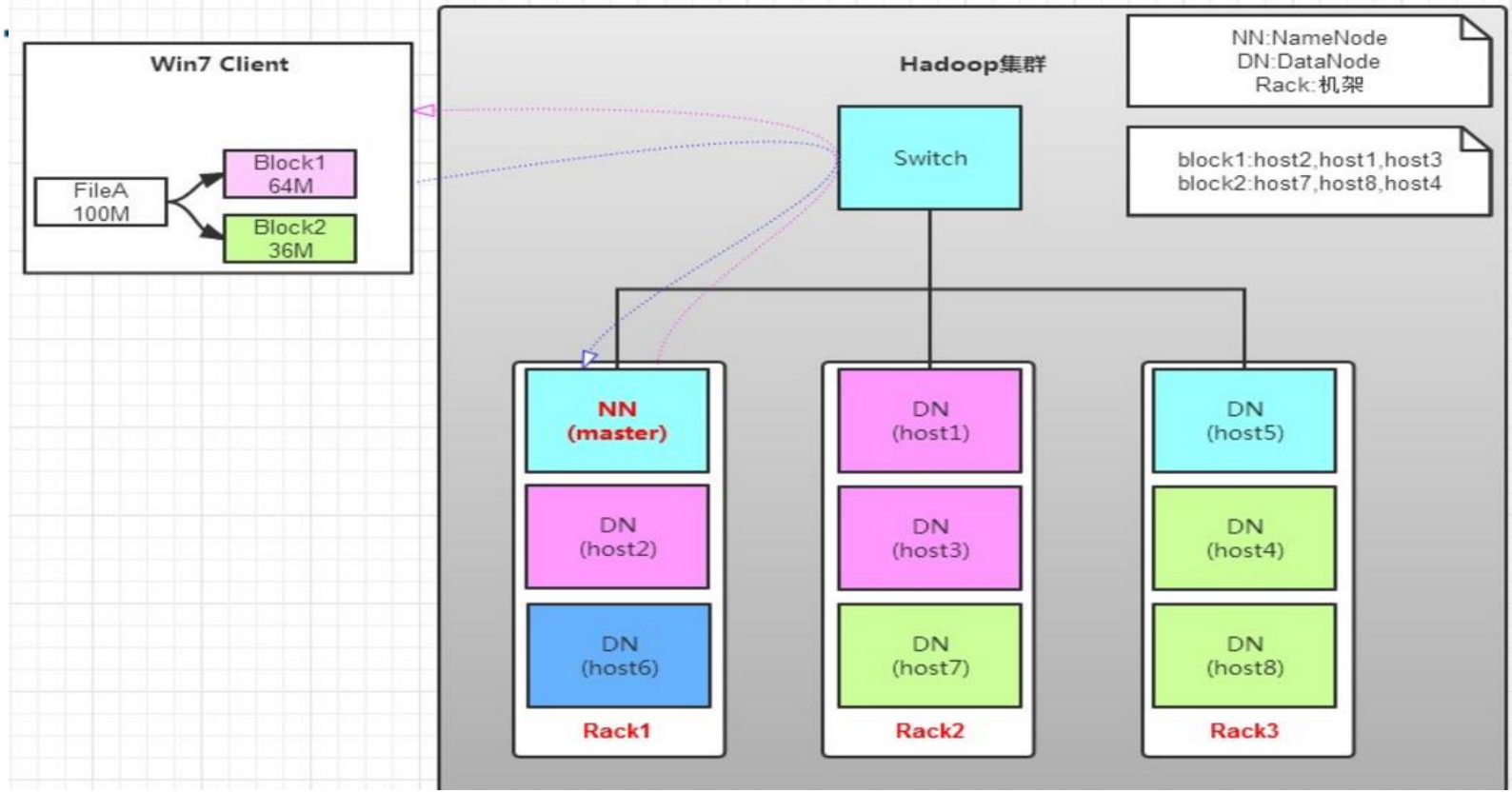

Block successfully stored and replicated in HDFS
When I am done with a block, I repeat the same steps with remaining blocks
WHEN ALL BLOCKS ARE WRITTEN..
All blocks written, please close file
Case closed !! NOW I store all meta information in persistent storage (hard disks)
RECAP
I divided the file in blocks--
--for each block, I provided address of datanodes--
--we stored data via Replication Pipeline

65

HDFS读过程


REQUEST FROM USER
Writing file in HDFS -- check. What about reading them? Let's ask the client again..
Mr. Client, please read this file for me..
Roger...
CONTACT NAMENODE FIRST..
Please give me info on this file
Filename
I reply (a) list of all blocks for this file, (b) list of datanodes for each block (sorted by distance from client)
Block 1: at DN x1, y1, z1
Block 2: at DN x2, y2, z2
Block 3: at DN x3, y3, z3
...and so on...
Now I know how many blocks to download, and the datanodes where each block is stored
So I download each block, in turn, like so --
DOWNLOAD DATA
Download data from the nearest datanode (the first in list)
Please give me block n
DATA for block n
Umm.. Question -- What happens when the datanode is dead, or does not have the data, or the data is corrupted ...
Actually, HDFS can very elegantly handle these faults and more as we will see next --

读操作实例2



a. client向namenode发送读请求。

b. namenode查看Metadata信息，返回fileA的block的位置。

block1:host2,host1,host3

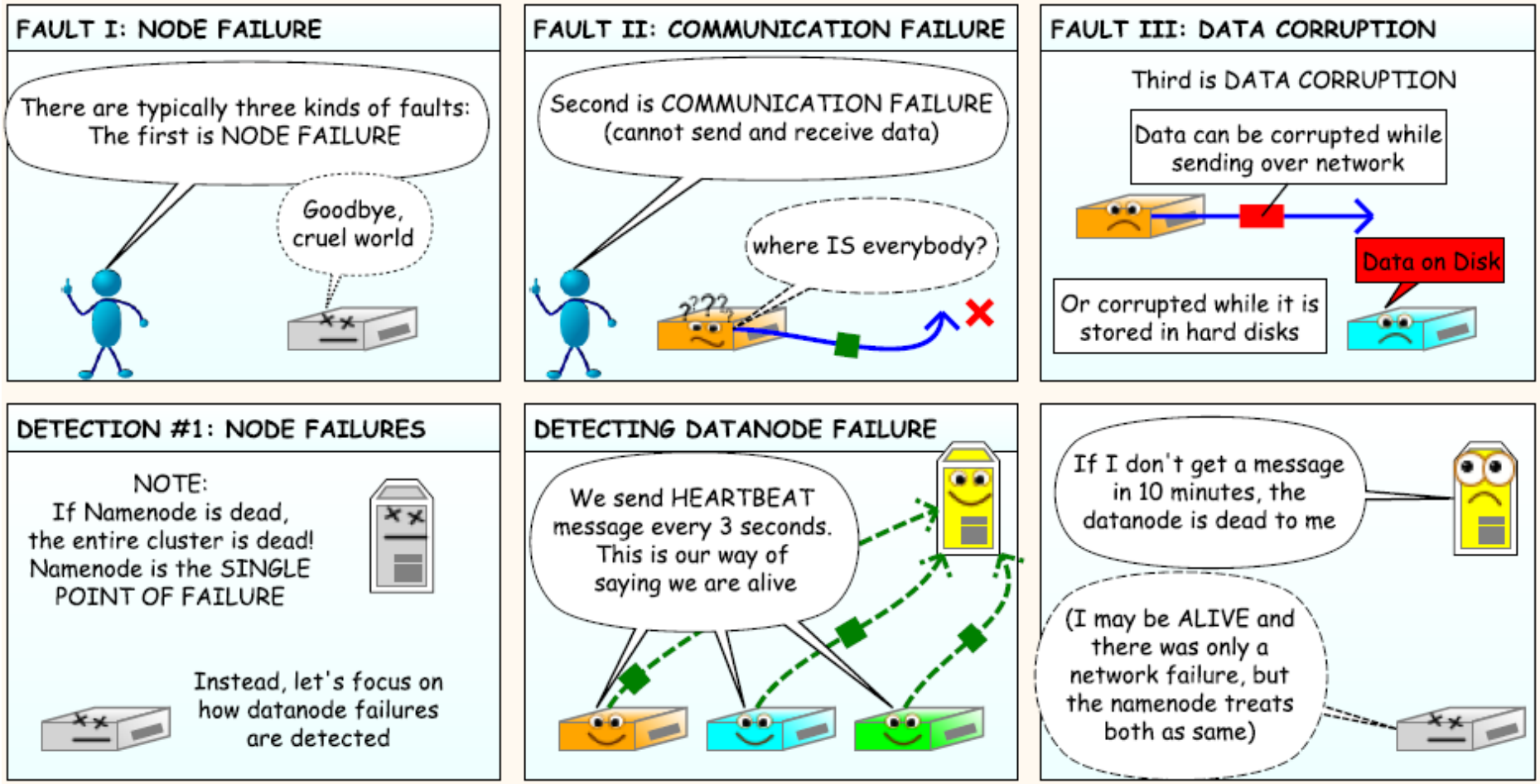
block2:host7,host8,host4

c. block的位置是有先后顺序的，先读block1，再读block2。而且block1去host2上读取；然后block2，去host7上读取；

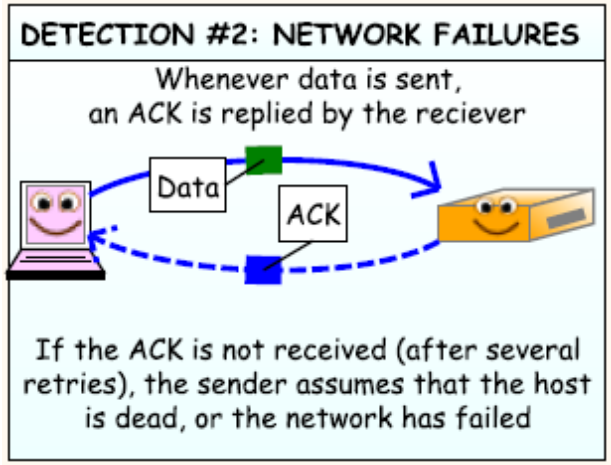
上面例子中，client位于机架外，那么如果client位于机架内某个DataNode上，例如,client是host6。那么读取的时候，遵循的规律是：

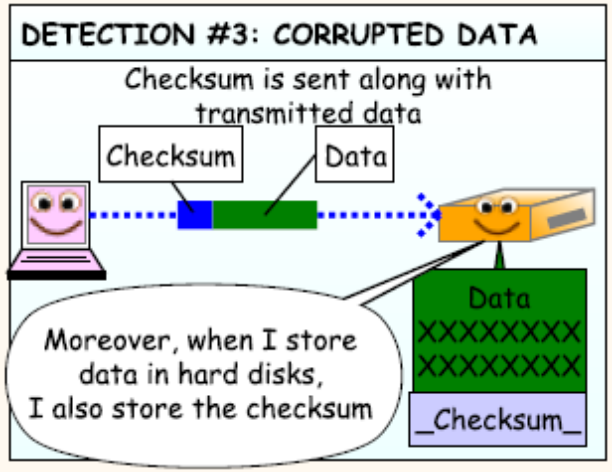
优选读取本机架上的数据。

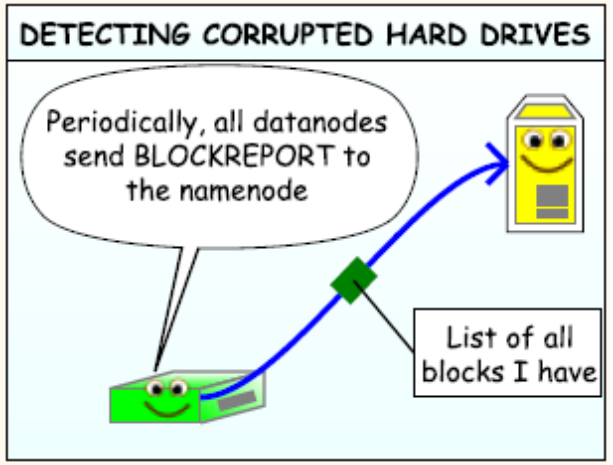
HDFS容错机制(1)-分类和检测

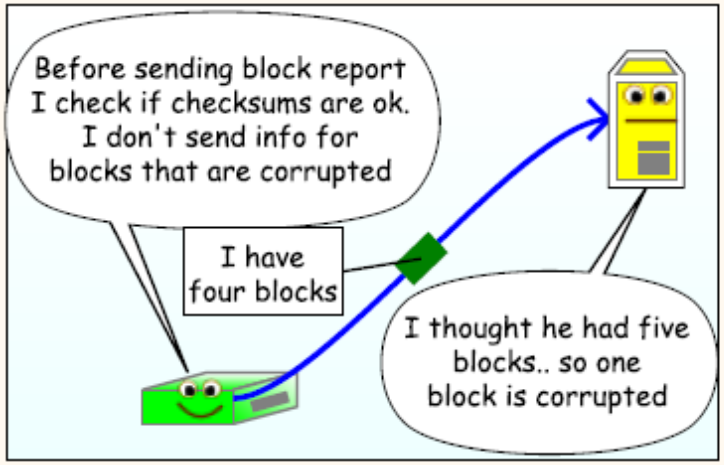


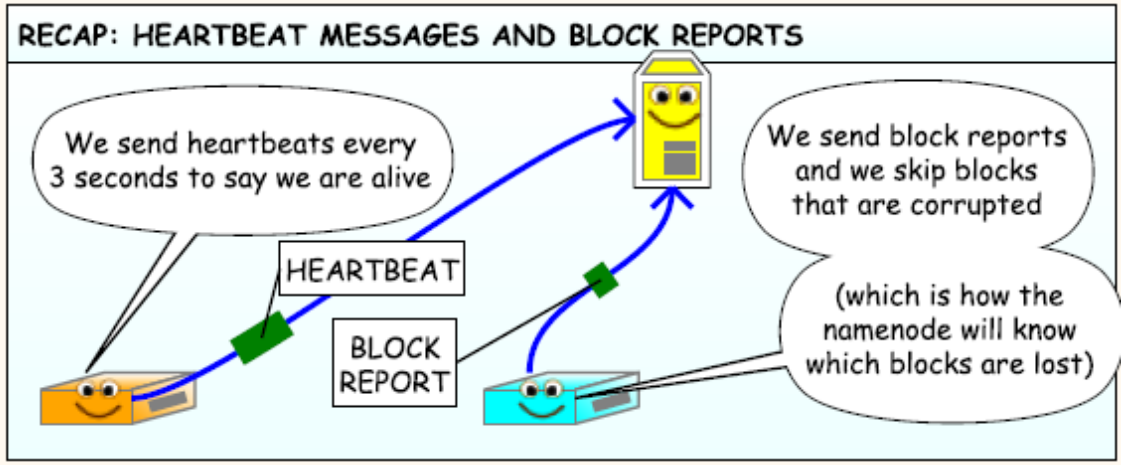
HDFS容错机制(1)-分类和检测



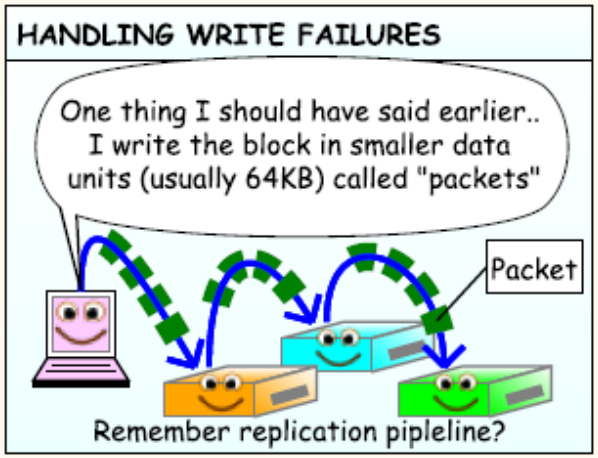


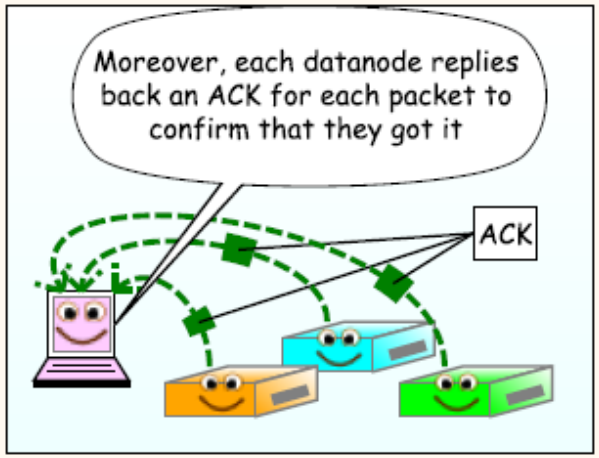


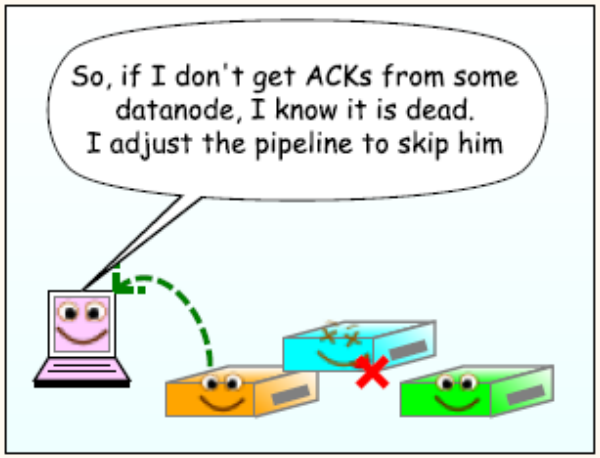


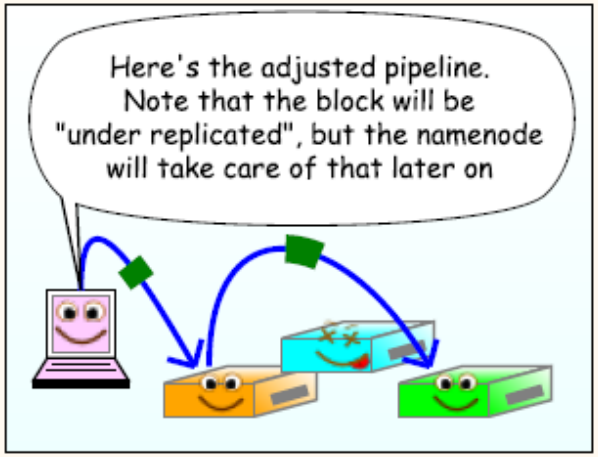


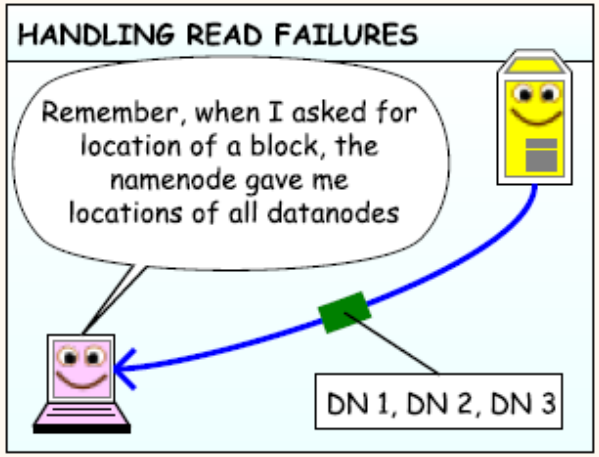
HDFS容错机制(2)-读写故障

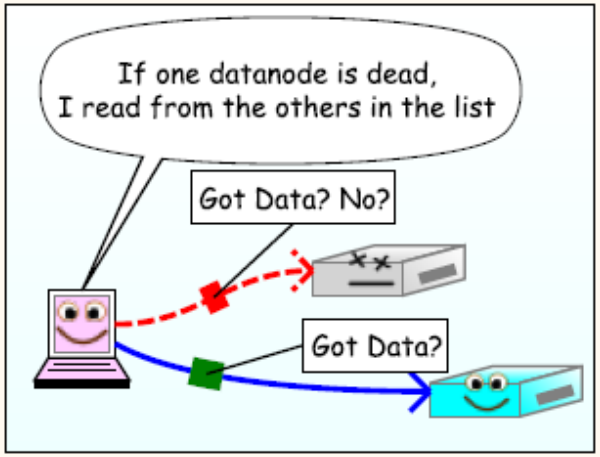




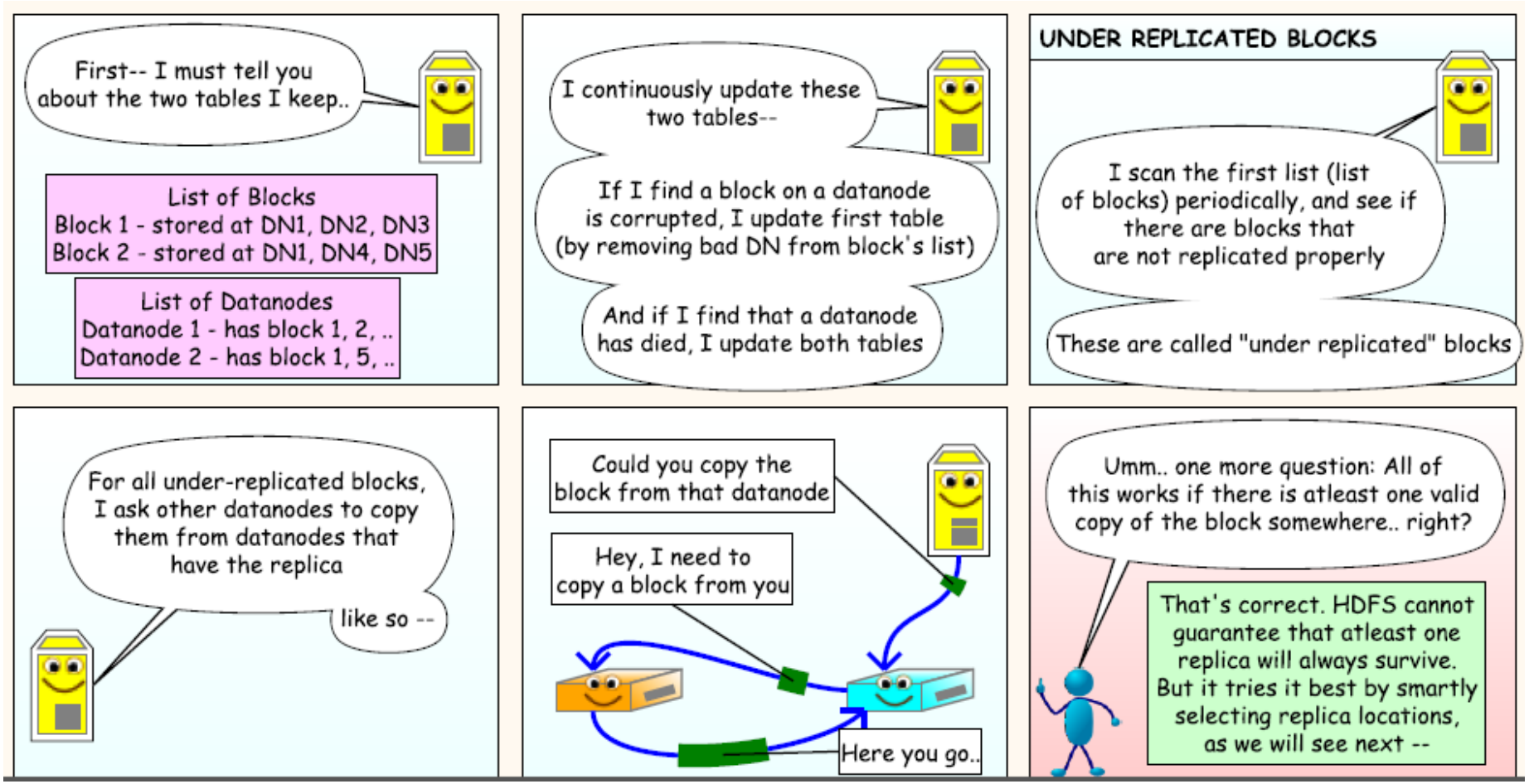






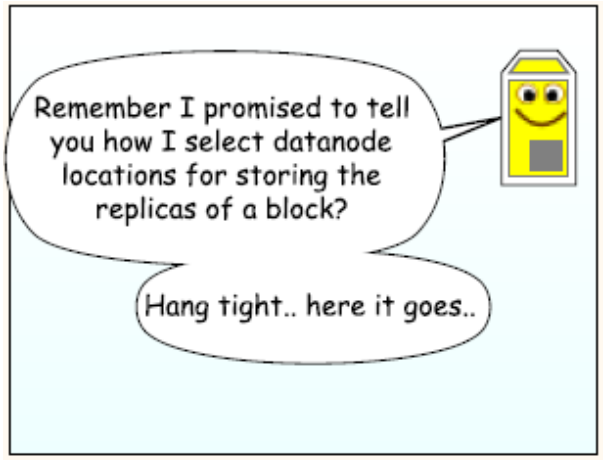


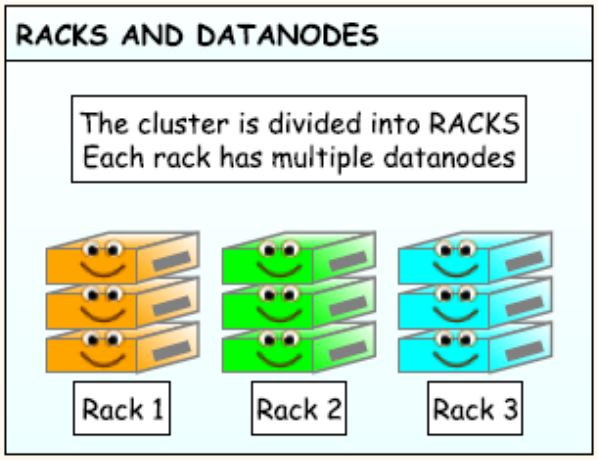
HDFS容错机制(3)-数据节点故障

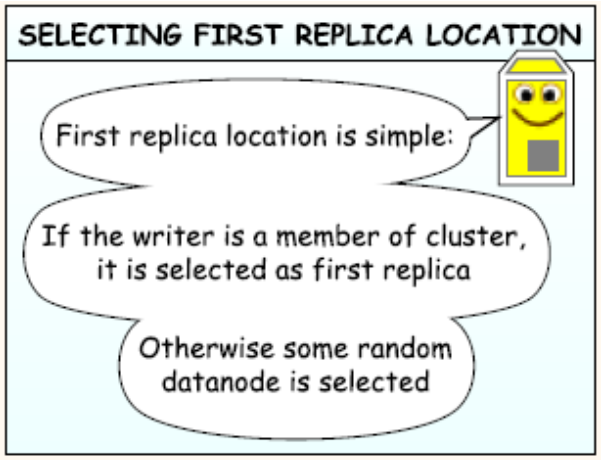


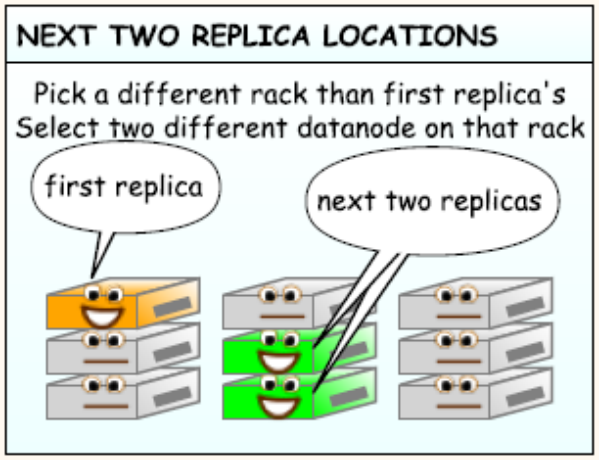
71

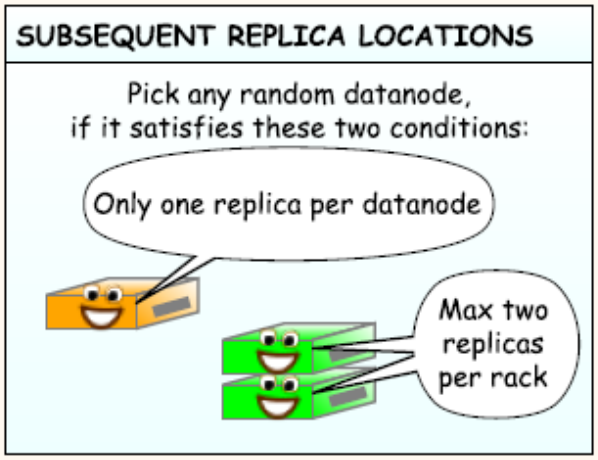
HDFS副本放置策略

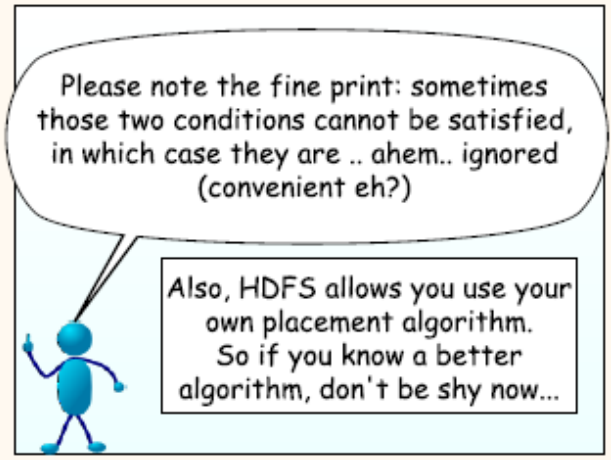

Remember I promised to tell you how I select datanode locations for storing the replicas of a block?
Hang tight.. here it goes..


RACKS AND DATANODES
The cluster is divided into RACKS
Each rack has multiple datanodes
Rack 1
Rack 2
Rack 3


SELECTING FIRST REPLICA LOCATION
First replica location is simple:
If the writer is a member of cluster, it is selected as first replica
Otherwise some random datanode is selected


NEXT TWO REPLICA LOCATIONS
Pick a different rack than first replica's
Select two different datanode on that rack
first replica
next two replicas


SUBSEQUENT REPLICA LOCATIONS
Pick any random datanode,
if it satisfies these two conditions:
Only one replica per datanode
Max two replicas per rack


Please note the fine print: sometimes those two conditions cannot be satisfied, in which case they are .. ahem.. ignored (convenient eh?)
Also, HDFS allows you use your own placement algorithm. So if you know a better algorithm, don't be shy now...

GFS和HDFS对比

	GFS	HDFS
实现架构	Master/slave（单一主控机+多台工作机）	Master/slave（单一主控机+多台工作机）
性质	分布式文件系统	分布式文件系统
高性能要求	通过数据分块和复制功能实现	通过数据分块和复制实现
追加操作处理	允许多客户端并发追加同一文件，存在重复追加和乱序问题	只允许一次打开并追加数据，利用临时文件操作解决乱序和重复记录的问题。
数据节点	有主副之分，存在Lease授权，减少追加时master的工作量	无主副之分，不存在Lease授权
Master单点失效处理	对master的数据有备份，master失效时，选举备机接替主master继续对外服务	Master的数据只写入到本机，master失效，需要人工介入
对快照的支持	通过内部的copy-on-write数据结构实现集群快照功能	不提供快照功能

73

Cloud Simulation Tools

Tools

- CloudSim
- CloudAnalyst
- GreenCloud
- iCanCloud
- MDCSim
- NetworkCloudSim
- WorkflowSim
- VirtualCloud
- iFogSim
- CloudSched
- …….

CloudSim

- Motivation
  - Provides a generalized and extensible simulation framework that enables modelling, simulation, and experimentation of emerging Cloud Computing Infrastructures and application services
  - CloudSim is developed in CLOUDS Lab directed by by Prof. Rajkumar Buyya, University of Melbourne
  - The most popular version released in 2012

CloudSim

- Main Features
  - CloudSim models data centers, virtual machines, service brokers, and resource provisioning methods
  - Provides a flexible switching between space-shared and time-shared allocation of processing elements to services
  - Researchers can implement cloud-based application by very less effort and time and test the performance in heterogeneous environments

CloudSim

- Many leading organizations and universities around the world are using CloudSim for:
  - Cloud resource provisioning
  - Energy-efficient management of data center resources
  - Optimization of cloud computing
  - Research activities
- Limitation: No Graphical User Interface (GUI)

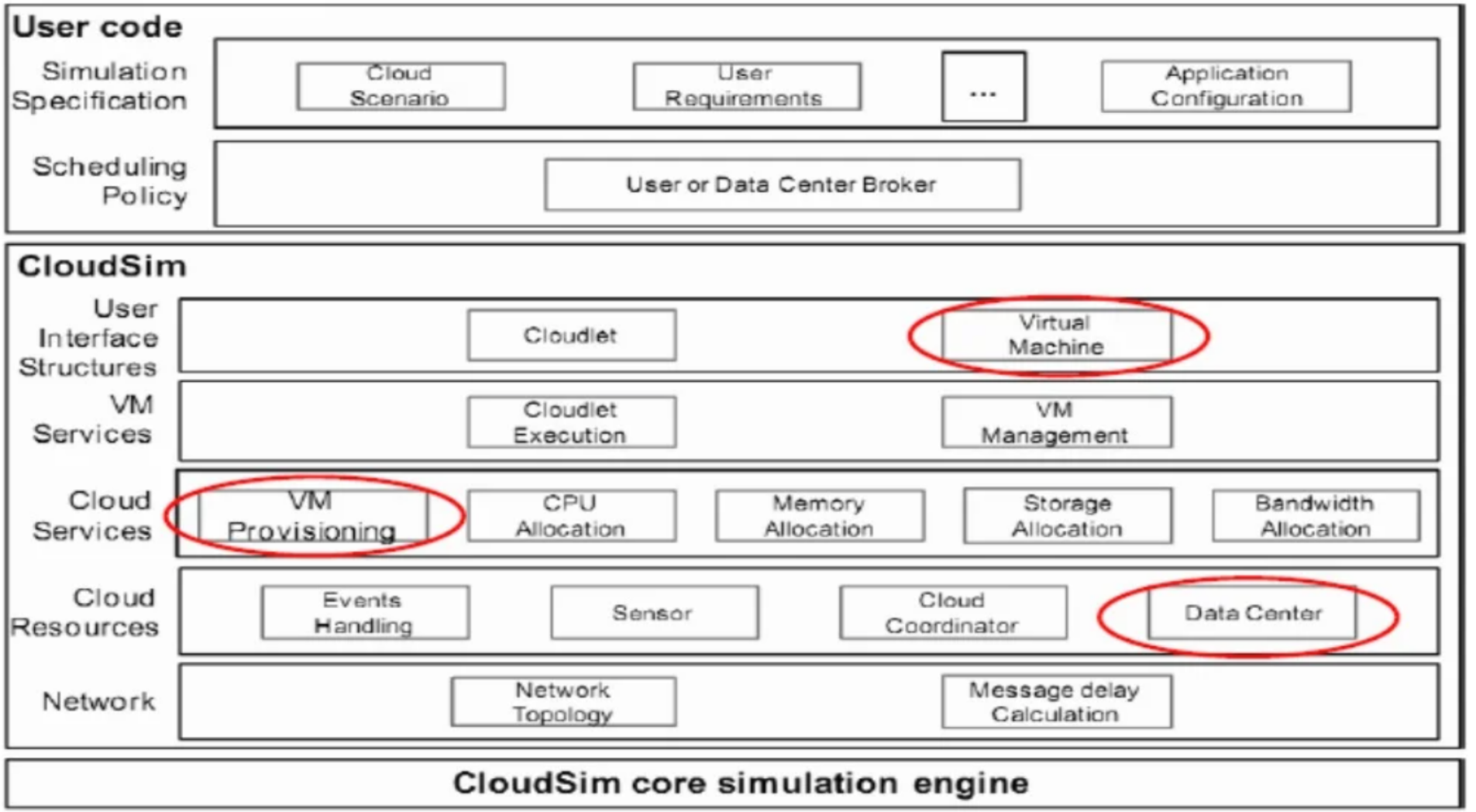


Figure 3. Layered CloudSim architecture.

CloudAnalyst

- Motivation:
  - A lack of tools that enable developers to evaluate requirements of large-scale Cloud applications in terms of geographical distribution of both computing servers and user workloads
  - Simulate large-scale Cloud applications with the purpose of studying the behavior of such applications under various deployment configurations

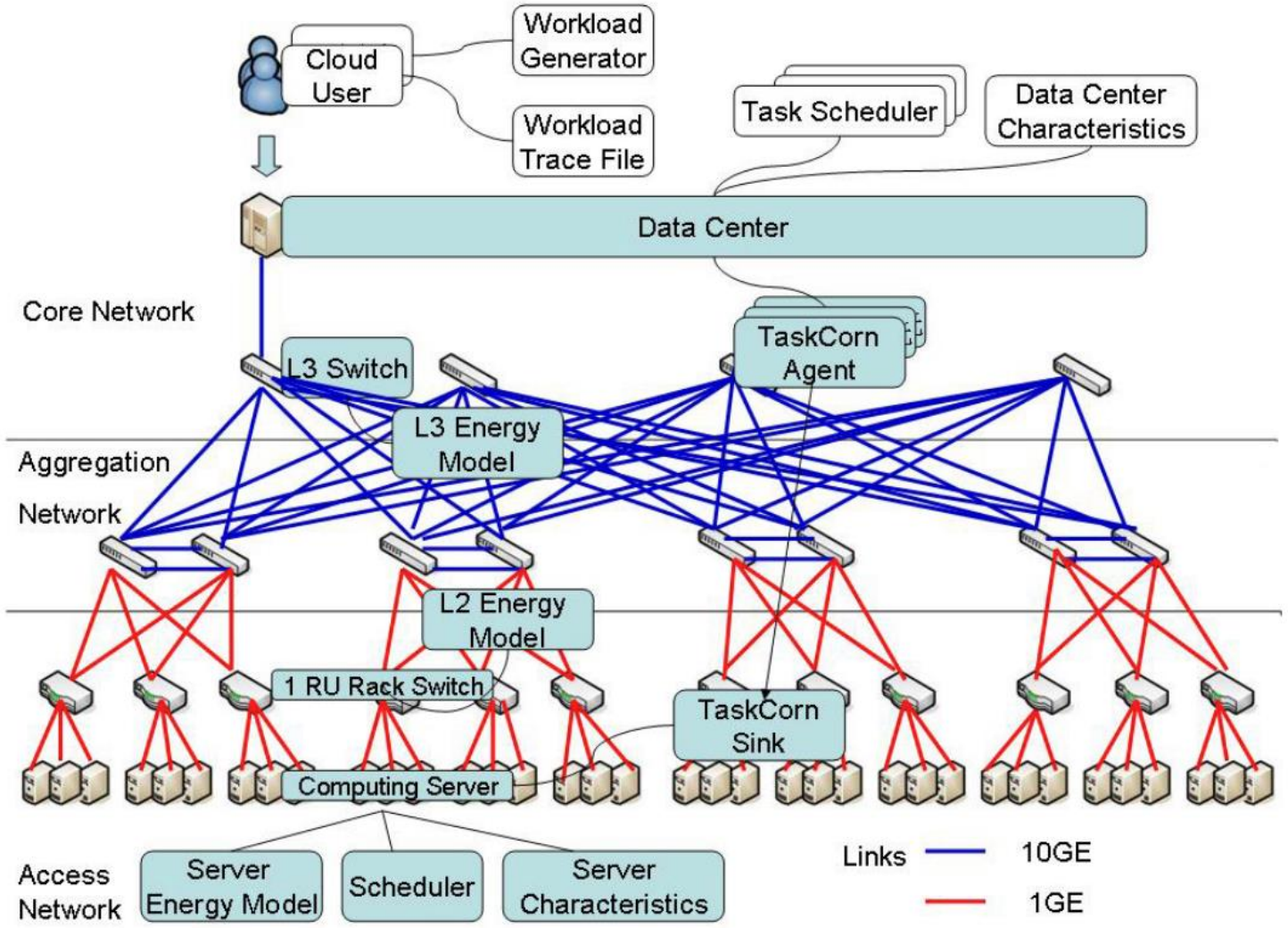
CloudAnalyst

- Main Features:
  - Easy to use GUI
  - Graphical output
  - Use of consolidated technology and ease of extension
  - Ability to define a simulation with a high degree of configurability and flexibility
  - Repeatability of experiments


Cloud Analyst
Help
Configure Simulation
Define Internet Characteristics
Run Simulation
Exit
DC1
UB1
Show Region Boundaries

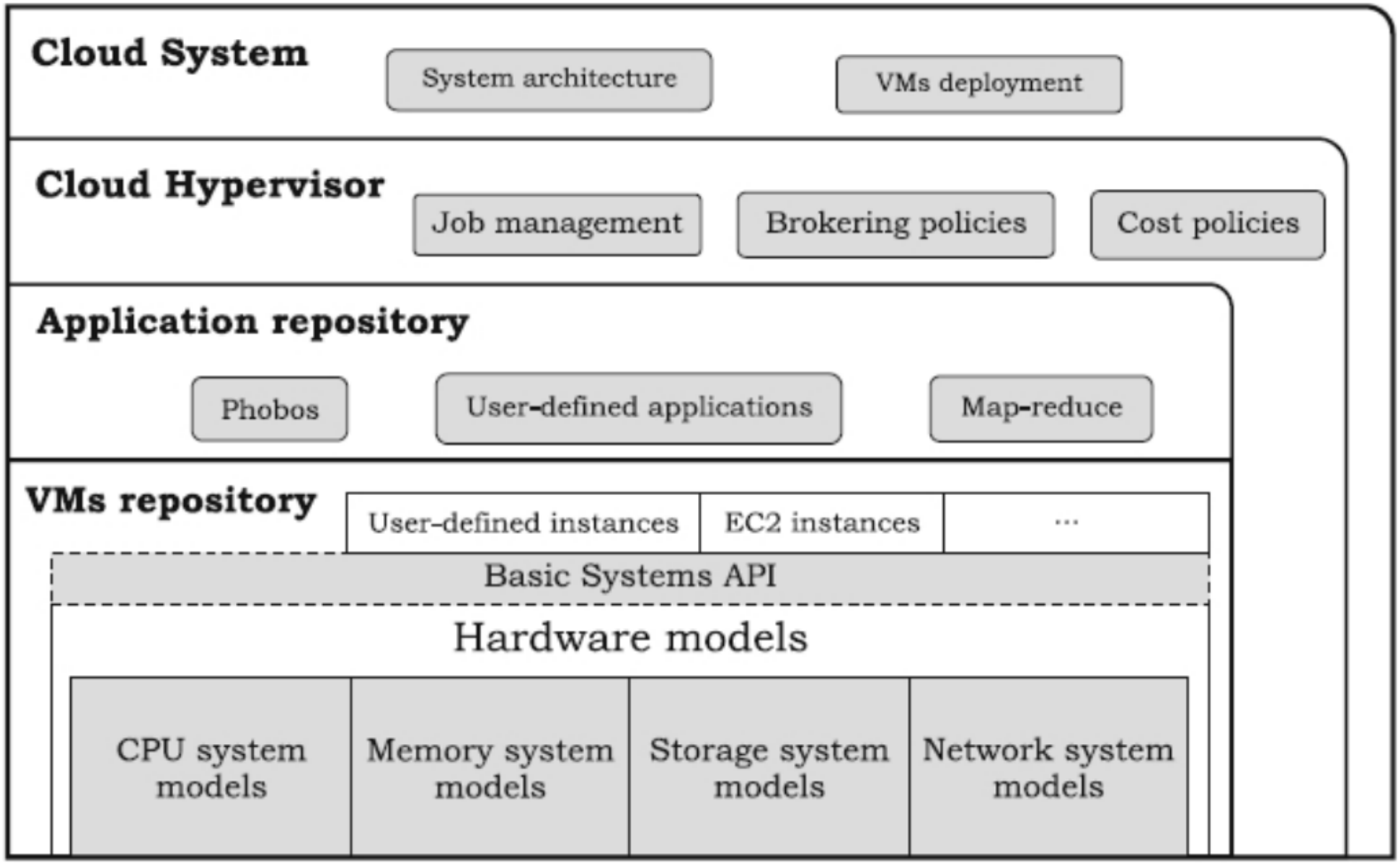
GreenCloud

- Motivation:
  - The lack of detailed simulators on the market
  - GreenCloud is a sophisticated open source cloud computing simulator
  - GreenCloud as been elaborated in the context of the GreenIT project
- Main Features:
  - Simulation environment for energy-aware cloud computing data centers
  - The extension of the well-known NS2 network simulator
  - Focused mainly on the communications within a cloud, i.e., all the communication processes are simulated at packet level


Cloud User
Workload Generator
Workload Trace File
Task Scheduler
Data Center Characteristics
Data Center
Core Network
L3 Switch
TaskCorn Agent
L3 Energy Model
Aggregation Network
L2 Energy Model
1 RU Rack Switch
TaskCorn Sink
Computing Server
Access Network
Server Energy Model
Scheduler
Server Characteristics
Links
10GE
1GE

iCanCloud

- Motivation
  - Predict the trade-offs between cost and performance of a given set of application executed in a specific hardware, and then provide to users useful information about the such costs
  - Simulate instances types provided by Amazon
  - Developed by a research group (ARCOS) from Spain
- Main Features:
  - Customizable VMs can be used to quickly simulate uni-core/multi-core systems.
  - User-friendly GUI to ease the generation of large distributed models


Cloud System
System architecture
VMs deployment
Cloud Hypervisor
Job management
Brokering policies
Cost policies
Application repository
Phobos
User-defined applications
Map-reduce
VMs repository
User-defined instances
EC2 instances
...
Basic Systems API
Hardware models
CPU system models
Memory system models
Storage system models
Network system models

Comparisons of Cloud Simulation Toolkits (SMPT' 2015)

Table 1
Comparison guideline.

Items	CloudSim [20]	GreenCloud [9]	iCanCloud [3]	CloudSched [25]
Platform	Any	NS2	OMNET, MPI	Any
Programming language	Java	C++/OTcl	C++	Java
Availability	Open source	Open source	Open source	Open source
Graphical support	Limited (Via CloudAnalyst)	N	N	Y
Physical models	N	Limited (Via Plug-in)	Y	Y
Models for public cloud	N	N	Y	Y
Parallel experiments	N	N	Y	N
Energy consumption	Y	Y	N	Y
Migration algorithms	Y	N	N	Y
Simulation time	Seconds	Tens of minutes	Seconds	Seconds
Memory space	Small	Large	Medium	Small

Comparisons of Cloud Simulation Toolkits (SMPT' 2015)

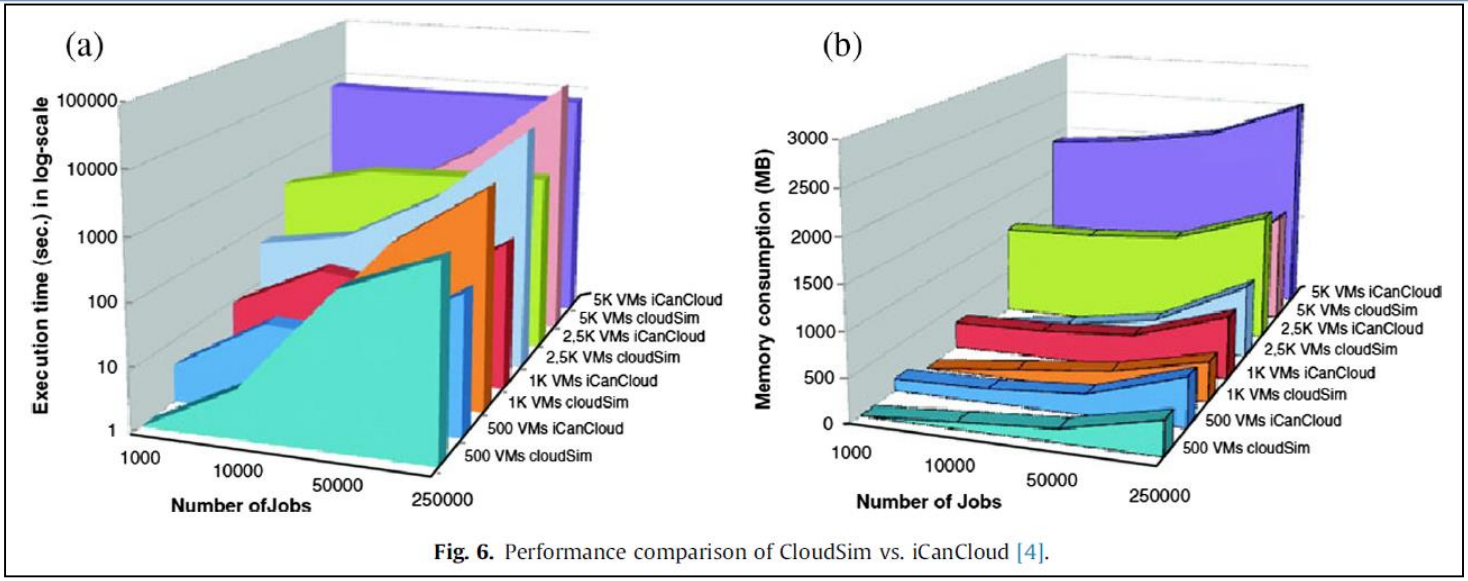


Fig. 6. Performance comparison of CloudSim vs. iCanCloud [4].

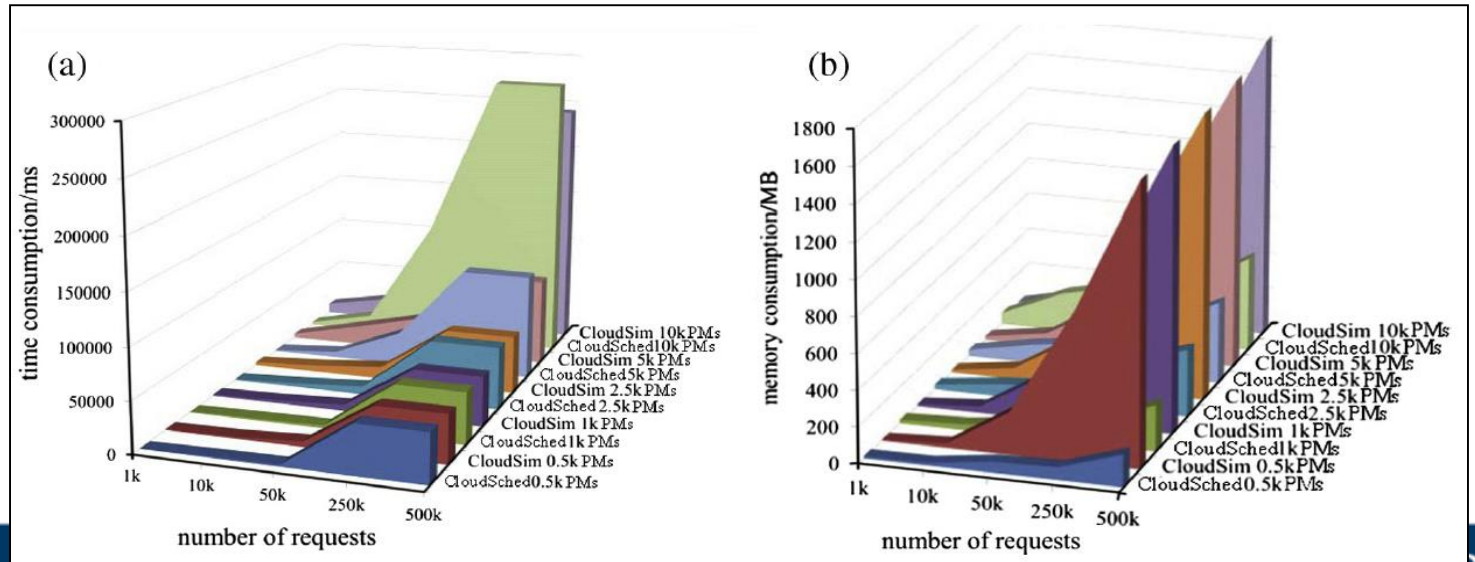


Fig. 7. Performance comparison of CloudSim vs. CloudSched.