Distributed and Parallel Technology

Datacenter, Warehouse and Cloud Computing

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

Semester 2 — 2016/17

[0]Based on earlier versions by Greg Michaelson and Patrick Maier

What Is Datacenter (Warehouse) Computing

A datacenter is

- a server farm — a room/floor/warehouse full of servers.
- *underpinning* the much wider area of *Cloud Computing*.

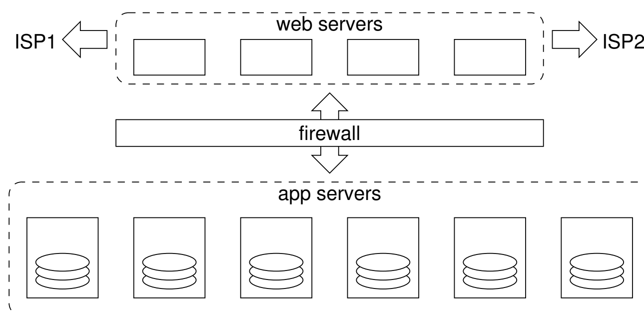
Datacenter requirements:

- Massive storage and compute power
- High availability — 24/7 operation, no downtimes acceptable
- High security against intrusion (both physical and electronic)
- Homogeneous hardware and software architecture
- Recently: Low power consumption

Who uses datacenters?

- Every big name on the internet: Google, Amazon, Facebook, ...
- Lots of other companies
  - to provide services over the internet and/or
  - to manage their business processes

Datacenter Computing — Architecture



Architecture stack:

- replicated web servers, connected to multiple ISPs
- firewall (often also replicated)
- 1000s of application servers, each with large disk
  - app servers run application software
  - data stored on app servers (via distributed FS/DB)
    - no dedicated file server/data base servers

Datacenter Computing — Application Programming

Problems:

- Distributed resources:
  - Often data distributed over 1000s of machines
- Performance:
  - Scalability of applications is paramount
- Availability:
  - Apps must cope with compute/network component downtimes
  - Apps must re-configure dynamically when cluster is upgraded

Solution: Distributed execution engine

- Offers skeleton-based programming model
- Hides most performance issues from programmer
  - Automatic parallelisation, controllable by few parameters
- Hides all distribution and availability issues from programmer
- Some particular engines:
  - MapReduce (Google; Apache Hadoop)
  - Dryad (Microsoft)

MapReduce — Overview

Observation

- Many applications conceptually simple (e.g. triv. data-parallel)
- Reliably dist. even simple apps on 1000s of nodes is a nightmare.
  - Skeletons could help!

MapReduce programming model

- MapReduce skeleton
  - Programmer writes only map and reduce functions.
  - Runtime system takes care of parallel execution and fault tolerance.

Implementation requires

- Distributed file system (Google: GFS, Hadoop: HDFS)
- For DB apps: dist. data base (Google: BigTable, Hadoop: HBase)

MapReduce — For Functional Programmers

What func programmers think when they hear "map/reduce"

```
-- map followed by reduce (= fold of associative m with identity e)
fp_map_reduce ::    (a -> b)
                -> (b -> b -> b) -> b
                -> [a] -> b
fp_map_reduce f m e = foldr m e . map f

-- map followed by group followed by groupwise reduce
fp_map_group_reduce ::    (a -> b)
                      -> ([b] -> [[b]])
                      -> ([b] -> b)
                      -> [a] -> [b]
fp_map_group_reduce f g r = map r . g . map f

-- list-valued map then group then groupwise list-valued reduce
fp_map_group_reduce' ::    (a -> [b])
                       -> ([b] -> [[b]])
                       -> ([b] -> [b])
                       -> [a] -> [[b]]
fp_map_group_reduce' f g r = map r . g . (concat . map f)
```

MapReduce — For Functional Programmers

Google's MapReduce (sequential semantics)

```
-- list-valued map then fixed group stage then list-valued reduce
map_reduce :: Ord c => ((a,b) -> [(c,d)])
                    -> (c -> [d] -> [d])
                    -> [(a,b)] -> [(c,[d])]
map_reduce f r = map (\ (k,vs) -> (k, r k vs)) .
                 (group . sort) .
                 (concat . map f)
  where sort :: Ord c => [(c,d)] -> [(c,d)]
        sort = sortBy (\ (k1,_) (k2,_) -> compare k1 k2)
        group :: Eq c => [(c,d)] -> [(c,[d])]
        group = map (\ ((k,v):kvs) -> (k, v : map snd kvs)) .
                groupBy (\ (k1,_) (k2,_) -> k1 == k2)
```

- Specialised for processing key/value pairs.
  - Group by keys
  - Reduction may depend on key and values
- Not restricted to lists — applicable to any container data type
  - Reduction should be associative+commutative in 2nd argument

MapReduce — Applications

TotientRange

```
-- Euler phi function
euler :: Int -> Int
euler n = length (filter (relprime n) [1 .. n-1])
  where relprime x y = hcf x y == 1
        hcf x 0 = x
        hcf x y = hcf y (rem x y)

-- Summing over the phi functions in the interval [lower .. upper]
sumTotient :: Int -> Int -> Int
sumTotient lower upper = head (snd (head (map_reduce f r input)))
    where input :: [((),Int)]
          input = zip (repeat ()) [lower, lower+1 .. upper]
          f :: ((),Int) -> [((),Int)]
          f (k,v) = [(k, euler v)]
          r :: () -> [Int] -> [Int]
          r _ vs = [sum vs]   -- reduction assoc+comm in 2nd arg
```

- Degenerate example: only single key
- Still exhibits useful parallelism
  - but would not perform well on Google's implementation

MapReduce — Applications

URL count

```
isURL :: String -> Bool
isURL word = "http://" `isPrefixOf` word

-- input:  lines of log file
-- output: frequency of URLs in input
countURL :: [String] -> [(String,[Int])]
countURL lines = map_reduce f r input
  where input :: [((),String)]
        input = zip (repeat ()) lines
        f :: ((),String) -> [(String,Int)]
        f (_,line) = zip (filter isURL (words line)) (repeat 1)
        r :: String -> [Int] -> [Int]
        r url ones = [length ones]
```

- Map phase
  1. breaks line into words
  2. filters words that are URLs
  3. zips URLs (which become keys) with value 1
- Group phase groups URLs with values (which = 1)
- Reduction phase counts #values

MapReduce — How To Parallelise

Sequential code

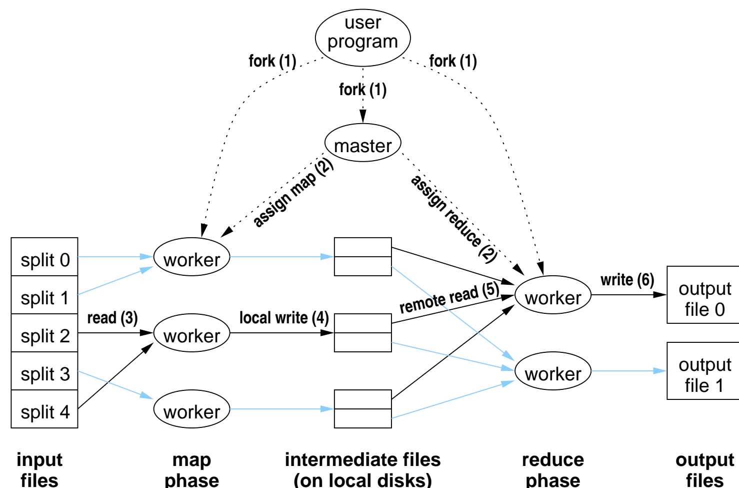
```
map_reduce f r = map (\ (k,vs) -> (k, r k vs)) .
                 (group . sort) .
                 (concat . map f)
```

suggests 3-stage pipeline

1. map phase
   - data parallel task farm
2. parallel sorting and grouping
   - parallel mergesort
3. groupwise reduce phase
   - data parallel task farm

Note: This is not how Google do it.

Google MapReduce — Execution Overview



- J. Dean, S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, Commun. ACM 51(1):107–113, 2008

Google MapReduce — Execution Overview

Execution steps:

1. User program forks master, M map workers, R reduce workers.
2. Master assigns map/reduce tasks to map/reduce workers.
   - Map task = 16–64 MB chunk of input
   - Reduce task = range of keys + names of M intermediate files
3. Map worker reads input from GFS and processes it.
4. Map worker writes output to local disk.
   - Output partitioned into R files (grouped by key)
5. Reduce worker gathers files from map workers and reduces them.
   1. Merge M intermediate files together, grouping by key.
   2. Reduce values groupwise.
6. Reduce worker writes output to GFS.
7. Master returns control to user program after all task completed.

Main Selling Points of MapReduce

- Easy to use for non-experts in parallel programming (details are hidden in the MapReduce implementation)
- Fault tolerance is integrated in the implementation
- Good modularity: many problems can be implemented as sequences of MapReduce
- Flexibility: many problems are instances of MapReduce
- Good scalability: using 1000s of machines at the moment
- Tuned for large data volumes: several TB of data
- Highly tuned parallel implementation to achieve eg. good load balance

Similar Datacenter Frameworks

Apache Hadoop

- MapReduce programming model
- Open source Java implementation
  - `http://hadoop.apache.org/`

Dryad (Microsoft)

- More general skeleton:
  - Programmer specifies directed acyclic dataflow graph:
    - vertex = sequential program
    - edge = unidirectional communication channel
  - Generalises Unix paradigm of "piping" apps together
- High-level languages building on Dryad
  - *Nebula* scripting language
  - *DryadLINQ*: database query compiler
- Closed source C++ implementation
- M. Isard et al. *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks*, EuroSys 2007

How MapReduce/Hadoop Work

Components:

- Distributed file system
  - We'll have a look at HDFS (Apache Hadoop)

    D. Borthakur. *HDFS Architecture*. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`
  - GFS (Google) is quite similar (but pre-dates MapReduce)

    S. Ghemawat, H. Gobioff, S. Leung. *The Google File System*. SOSP 2003
- Fault-tolerant task scheduling system
  - Can take advantage of distributed file system.

Hadoop HDFS — Architecture

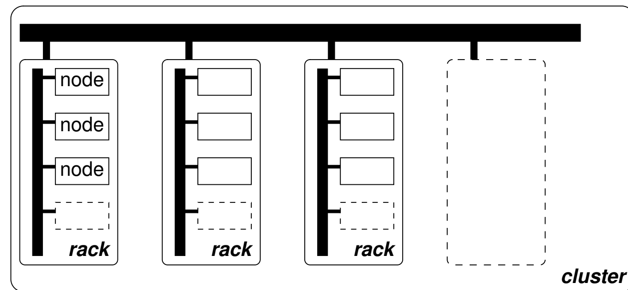
Design goals:

- Fault-tolerance
  - HDFS runs on large clusters. Hardware failures are common.
- Large files
  - Up to terabytes per file.
- Performance
  - Exploiting network locality where possible.
- Portable across heterogeneous hardware and software platforms.

Some design choices:

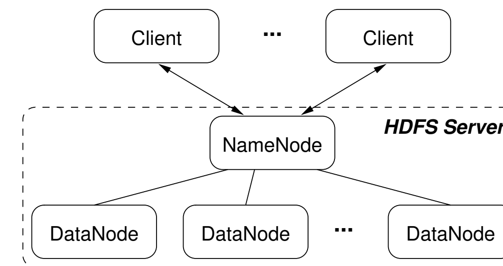
- Emphasis on high throughput rather than low latency
  - Optimise for streaming access rather than random access.
- Simple coherency model: 1 streaming writer / many readers.
  - Lock-free reading and writing.
- Processing facilities near the data.
  - Moving computations cheaper than moving large data sets.

Hadoop HDFS — Datacenter Network Topology



- Cluster consists of multiple racks.
- Inter-rack traffic needs to cross (at least) 2 more switches than intra-rack traffic.
  - inter-rack latency $>$ intra-rack latency
  - aggregate intra-rack bandwidth $\gg$ inter-rack bandwidth
- HDFS must optimise for parallel intra-rack access where possible.

Hadoop HDFS — Logical Architecture



- Client/Server architecture (with distributed server)
- HDFS server implements master/worker pattern:
  - 1 master: NameNode (stores HDFS meta-data on local FS)
  - Many workers: DataNodes (store HDFS data on local FS)
- Client applications may run on DataNodes (rather than on separate machines).

Hadoop HDFS — File Organisation

Metadata (on NameNode)

- Hierarchical namespace (file/directory tree)
- Limited access control
  - File system access control not important if clients are trusted.

Data (on DataNodes)

- Files stored in large blocks spread over the whole cluster.
  - Block size up to 64 MB (configurable per file)
  - Each block stored as a file on DataNode's local FS
- Each block replicated n times.
  - Replication factor n configurable per file (default $n = 3$)
  - Replicas spread over the cluster
    1. to achieve high availability, and
    2. to increase performance.

  Replication strategy crucial for both objectives.

Hadoop HDFS — Replication Strategies

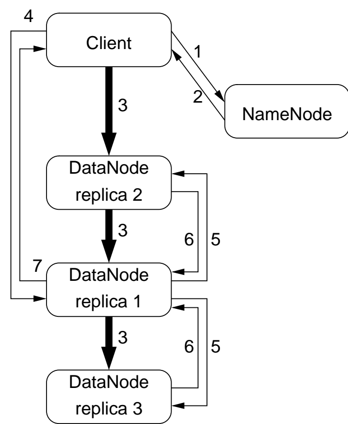
Two simple replication strategies:

1. Spread all replicas evenly over racks of cluster.
   - \+ Optimises availability.
   - – Performance may suffer if clients have to fall back on off-rack replicas.
2. Retain one replica on same rack as primary replica, spread remaining replicas over other racks.
   - \+ Optimises performance (even primary replica unavailable).
   - – Availability almost as high as with strategy 1.
     - ★ Single node failures more common than full rack failures.

Many more strategies are possible.

- Finding good (high availability + high performance) strategies is a hot research topic in distributed file systems / data bases.

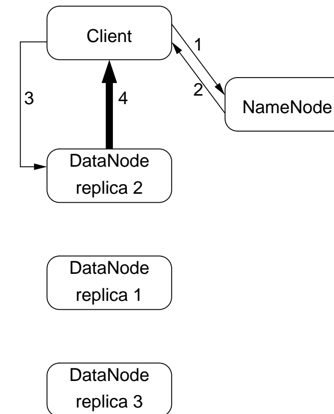
Hadoop HDFS — Writing to a File



1. Request to write to block *i* in file *F*
2. Return block handle and locations
3. Data transfer/forwarding
   - Client sends data to nearest replica.
   - Replicas form forwarding pipeline.
4. Initiate write on primary replica
5. Initiate replication
6. Commit secondary replica
7. Commit write
   - Commit only if writing successful on all replicas.
   - Otherwise return error msg to client.

Note: Minimal interaction with NameNode (to avoid bottleneck).

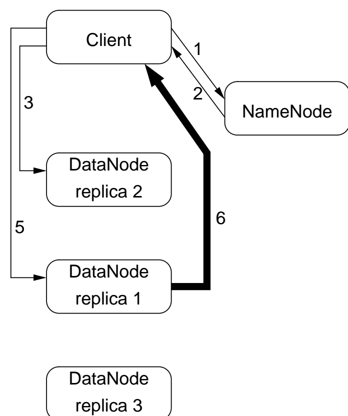
Hadoop HDFS — Reading from a File



1. Request to read from block *i* in file *F*
2. Return block handle and locations
   - Client may cache handle and locations.
     - ★ Only cache for short time: location data becomes invalid quickly.
3. Request data
   - Client requests data from nearest replica.
4. Transfer data

Note: Minimal interaction with NameNode.

Hadoop HDFS — Reading from a Dead Node

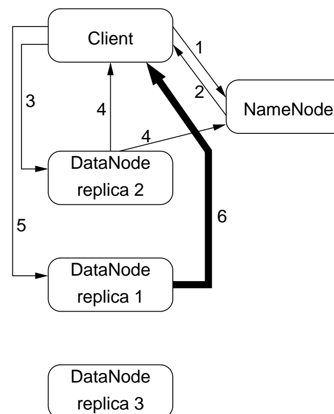


1. Request to read from block *i* in file *F*
2. Return block handle and locations
3. Request data
   - Client requests data from nearest replica.
4. Request times out because
   - network down,
   - DataNode dead or
   - DataNode overloaded.
5. Re-request data
   - Client requests data from another replica.
6. Transfer data

Note: No additional interaction with NameNode.

- NameNode will detect and deal with failure independently.

Hadoop HDFS — Reading from a Corrupt File



1. Request to read from block *i* in file *F*
2. Return block handle and locations
3. Request data
4. Signal corruption
   - Inform both Client and NameNode.
5. Re-request data
   - Client requests data from another replica.
6. Transfer data

Note: No additional Client/NameNode interaction.

- NameNode will deal with failure independently.

Hadoop HDFS — Fault Tolerance

Fast recovery from NameNode failure

- Snapshot of NameNode's state regularly dumped to disk and replicated off-rack.
- State updates between snapshots journaled.
- **Note:** No automatic NameNode failure detection.

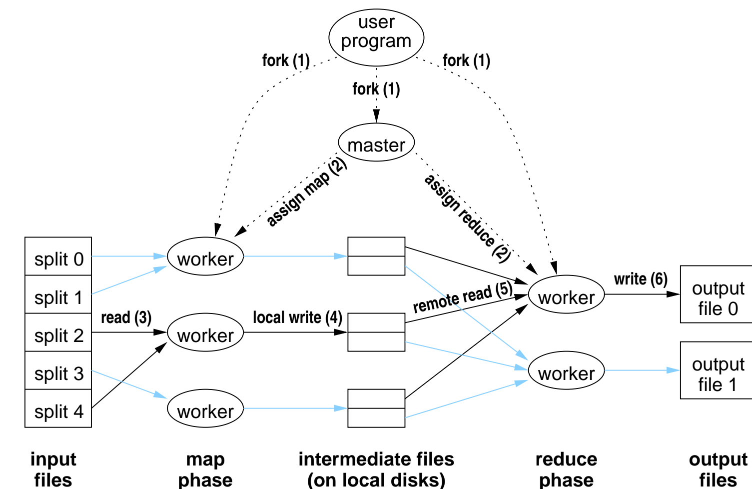
Recovery from DataNode failure or network partition

- Failure detection: NameNode listens to DataNodes' heartbeats.
- Fast recovery:
  1. Clients timeout and fall back on other replicas.
  2. NameNode initiates re-replication of blocks residing on dead nodes.

Recovery from data corruption

- Failure detection: DataNode compares checksums.
- Fast recovery:
  1. Clients timeout and fall back on other replicas.
  2. DataNode notifies NameNode, which initiates re-replication of corrupt block.

How MapReduce Piggybacks on HDFS



How MapReduce Piggybacks on HDFS

Fault-tolerance:

- MapReduce worker heartbeat monitoring done by HDFS.
  - DataNodes run Map or Reduce workers as applications.
  - NameNode runs MapReduce master as application.
    - If DataNode dead, re-replicate its data and re-execute its Map or Reduce tasks somewhere else.

Performance:

- Select Map workers optimising for local reads.
  - Prefer DataNodes with blocks of input file on disk.
  - Failing that, prefer DataNodes with blocks of the input file on the same rack.
- Have Reduce worker write primary replica to same DataNode.
  - Replication strategy distributes output file evenly over other racks.
- Select Reduce optimising access to all Map workers.
  - Prefer grouping Reduce workers on same rack as Map workers.

Further Reading:

- Tom White, *"Hadoop: The Definitive Guide"*. O'Reilly Media, Third edition, May 2012.
- Hadoop documentation: `http://hadoop.apache.org/`
- See also the links on Hadoop-level "scripting" languages such as Pig and Hive.
- D. Borthakur, *"HDFS Architecture"*. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`