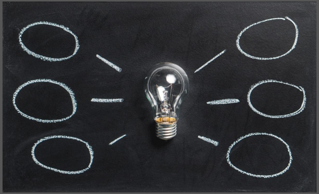
Department of Computer Science

University of Reading

Designing of Distributed Systems



Julian Kunkel

2019-11-

Learning Objectives

- Sketch various distributed architectures
- Describe example problems and algorithms for distributed systems
- Discuss semantics and limitations when designing distributes systems
- Explain the CAP-theorem
- Design systems using the RESTful architecture

Outline

Outline

Components for Big Data Analytics

Required components for a big data system

- Servers, storage, processing capabilities
- User interfaces

Storage

- NoSQL databases are non-relational, distributed and scale-out
  - Hadoop Distributed File System (HDFS)
  - Cassandra, CouchDB, BigTable, MongoDB[1]
- Data Warehouses are useful for well known and repeated analysis

Processing capabilities

- Interactive processing is difficult
- Available technology offers
  - Batch processing (hours to a day processing time)
  - "Real-time" processing (seconds to minutes turnaround)

Basic Considerations for Storing Big Data

Analysis requires efficient (real-time) processing of data

- New data is constantly coming (Velocity of Big Data)
  - How do we technically ingest the data?
    - In respect to performance and data quality
  - How can we update our derived data (and conclusions)?
    - Incremental updates vs. (partly) re-computation algorithms
- Storage and data management techniques are needed
  - How do we map the logical data to physical hardware and organize it?
  - How can we diagnose causes for problems with data (e.g., inaccuracies)?

Alternative Processing Technology

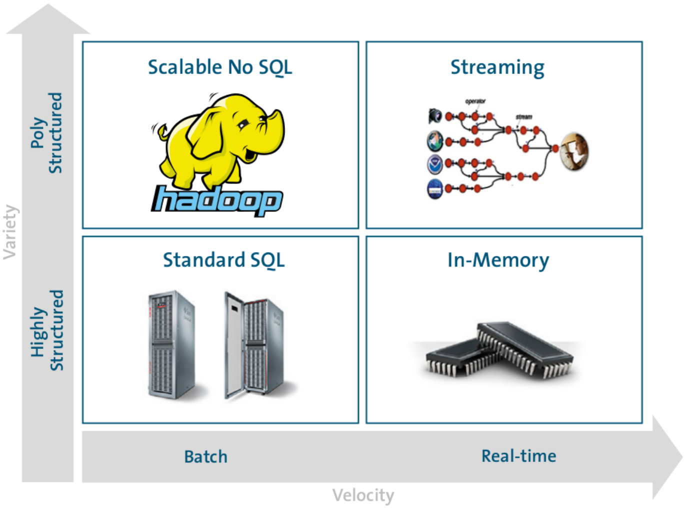


Figure: Source: Forrester Webinar. Big Data: Gold Rush Or Illusion? [4]

The Lambda Architecture [11]

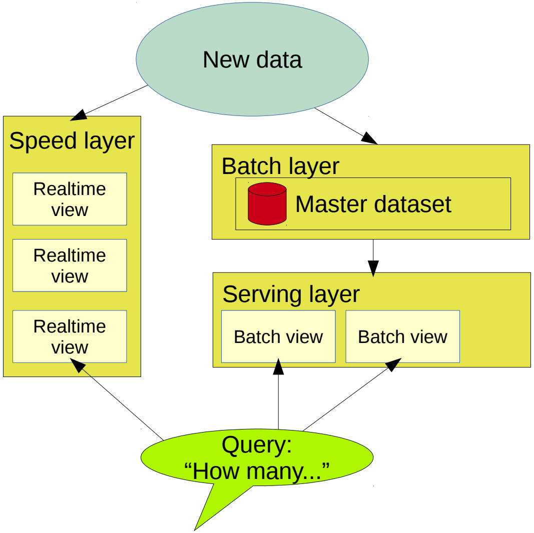


Figure: Redrawn figure. Source: [11], Fig. 2.1

- Goal: Interactive Processing
- Batch layer pre-processes data
  - Master dataset is immutable/never changed
  - Operations are periodically performed
- Serving layer offers performance optimized views
- Speed layer serves deltas of batch and recent activities, may approximate results
- Robust: Errors/inaccuracies of realtime views are corrected in batch view

Outline

How to Write an Algorithm: Programming Paradigms [14]

Programming paradigms: process models [15] for computation

- Fundamental style and abstraction level for computer programming
  - **Imperative** (e.g., Procedural)
  - **Declarative** (e.g., Functional, **Dataflow**, Logic)
  - **Data-driven** programming (describe patterns and transformations)
  - **Multi-paradigm** support several at the same time (e.g., **SQL**)
- Goals: productivity of the users and performance upon execution
  - Tool support for development, deployment and testing
  - Performance depends on single core efficiency but importantly parallelism
- **Parallelism** is an important aspect for processing
  - In HPC, there are language extensions, libraries to specify parallelism
    - PGAS, Message Passing, OpenMP, data flow e.g., OmpSs, ...
  - In BigData Analytics, libraries and domain-specific languages
    - MapReduce, SQL, data-flow, streaming and data-driven

Semantics

Semantics describe operations and their behavior, i.e., the property of the service

- Application programming interface (API)
- **Concurrency**: Behavior of simultaneously executed operations
  - Atomicity: Are partial modifications visible to other clients
  - Visibility: When are changes visible to other clients
  - Isolation: Are operations influencing other ongoing operations
- **Availability**: Readiness to serve operations
  - Robustness of the system for typical (hardware and software) errors
  - Scalability: availability and performance behavior depending on the number of clients, concurrent requests, request size, ...
  - Partition tolerance: Continue to operate even if network breaks partially
- **Durability**: Modifications should be stored on persistent storage
  - Consistency: Any operation leaves a consistent (correct) system state

Wishlist for Distributed Software

- High-availability, i.e., you can use the service all the time
- Fault-tolerance, i.e., can tolerate errors
- Scalable, i.e., the ability to be used in a range of capabilities
  - Linear scalability in respect to data volume
    - i.e., 2n servers handle 2n the data volume + same processing time
- Extensible, i.e., easy to introduce new features and data
- Usability: high user productivity
  - simple programming models
  - Real-time/interactive capabilities
    - The user can interact with the system without noticing delay
- Debuggability
  - In respect to coding errors and performance issues
- Little resource demand (compute and storage)
- Ready for the cloud

Consistency Limitations in Distributed Systems

CAP-Theorem

- It initially discusses implications, if the network is partitioned[2]
  - Consistency (here: visibility of changes among all clients)
  - Availability (we'll receive a response for every request)
  - Any technology can only achieve either consistency or availability

$\Rightarrow$ It is impossible to meet the attributes together in a distributed system:
  - Consistency
  - Availability
  - Partition tolerance (system operates despite network failures)

- **GroupWork (5 min): Discuss with a peer why they cannot be met together**
  - The proof can be found here `https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/`

Example Semantics

ACID

- Strict semantics for database systems to prevent data loss
- Atomicity, consistency, isolation and durability for **transactions**

BASE

- BASE is a typical semantics for Big Data due to the CAP theorem
- Basically Available replicated Soft state with Eventual consistency [26]
  - Availability: Always serve but may return a failure, retry may be needed
  - Soft state: State of the system may change over time without requests due to eventual consistency
  - Consistency: If no updates are made any more, the last state becomes visible to all clients after some time (eventually)
- Big data solutions usually exploit the immutability of data

POSIX Input/Output of Data (read/write)

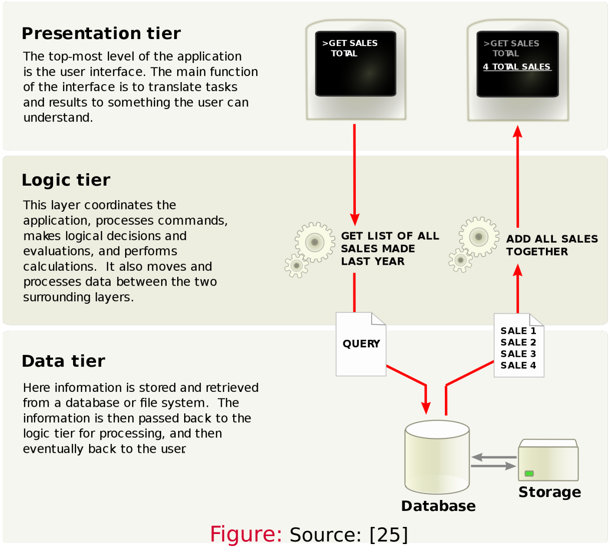
- Atomicity and isolation for individual I/O operations, locking possible

Architectural Patterns for Distributed Systems [19]

Architectural patterns provide useful blueprints for structuring distributed systems

- Client-server: server provides service/functionality, client requests
- **Multilayered** architecture (n-tier): separating functionality
  - 3-tier separates: presentation, application processing, data management
- Peer-to-peer: partition workload between equipotent/equal peers
- Shared nothing architecture: no sharing of information between servers
  - i.e., each server can work independently
- **Object request broker**: middleware providing transparency to function execution
  - Thus, the user invoking a function doesn't know where it is executed
  - The broker makes the decision where it is executed
  - *Remote Procedure Calls* (RPCs) are executed remotely
- Service-oriented architecture (SoA) encapsulates a discrete unit of functionality
  - Microservices: collection of loosely coupled service, lightweight protocols
- **Representational state transfer (REST)**: discussed later as an example

Multitier architecture [25]



Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Database
Storage

Figure: Source: [25]

Object Request Broker [24]

- Example: Common Object Request Broker Architecture (CORBA)
  - Example of the distributed object paradigm: object's appear local but are anywhere
  - Enables communication of systems that are deployed on diverse platforms, OS, programming languages, hardware
  - An OMG standard
- Remote method invocation (RMI)
- Interface Definition Language (IDL)
- Generation of "Stubs" for client and server
- Broker can forward requests to any servant

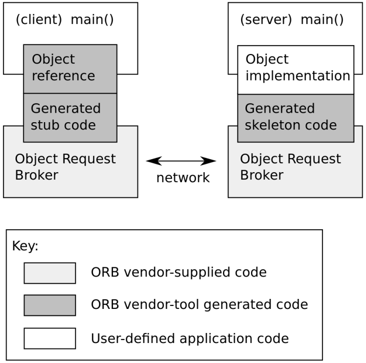


Figure: Example code. Source: Alksentrs, [24]

Outline

Problems and Standard Algorithms [20]

- Reliable broadcast: share information across processes
- Atomic commit: operation where a set of changes is applied as a single operation
- Consensus: distributed system agrees on a common decision
- Leader election: choose a single process to lead the distributed system
- Mutual exclusion: establish a distributed critical section; only one process enters
- Non-blocking data structures: provide global concurrent modification/access
- Replication: replicate data/information in a consistent way
- Resource allocation: provision/allocate resources to tasks/users
- Spanning tree generation

A Typical Problem: Consensus [17]

- **Consensus**: several processes agree (decide) for a single data value
  - Assume: Processes may propose a value (any time)
- Consensus and consistency of distributed processes are related
- Consensus protocols such as Paxos ensure cluster-wide consistency
  - They tolerate typical errors in distributed systems
  - Hardware faults and concurrency/race conditions
  - **Byzantine protocols** additionally deal with forged (lying) information
- Properties of consensus
  - **Agreement**: Every correct process must agree on the same value
  - **Integrity**: All correct process decide upon at most one value v. If one decides v, then v has been proposed by some process
  - **Validity**: If all process propose the same value v, then all correct processes decide v
  - **Termination**: Every correct process decides upon a value

Assumptions for Paxos

Requirements and *fault-tolerance assumptions* [16]

- Processors
  - **do not collude, lie, or otherwise attempt to subvert the protocol**
  - *operate at arbitrary speed*
  - *may experience failures*
  - *may re-join the protocol after failures (when they keep data durable)*
- Messages
  - **can be send from one processor to any other processor**
  - **are delivered without corruption**
  - *are sent asynchronously and may take arbitrarily long to deliver*
  - *may be lost, reordered, or duplicated*

Fault tolerance

- With 2F+1 processors, F faults can be tolerated
- With dynamic reconfiguration more, but $\leq$ F can fail simultaneously

Distributed Transactions [21]

- Goal: Atomic commitment of changes (e.g., transactions for databases)
- Consider the example of an order that requires to change several tables
- User order must update:
  - Customer information
  - Order table
  - (product table: item count)
- Assume the DB is distributed, e.g.,
  - Tables are on different hosts
  - Table keys are distributed
- How can we perform a safe commit?
  - With ACID semantics!
  - Either all operations or none complete

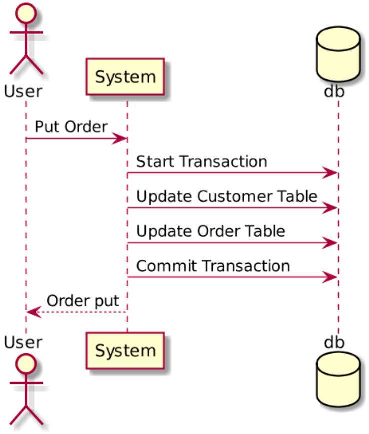


Figure: Execution UML; Source: [21]

Microservice Architecture for Bank Example [21]

- An architecture when splitting tables

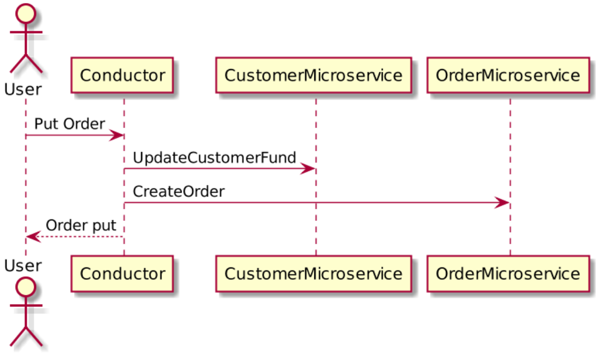


Figure: Source: [21]

Two-Phase Commit Protocol (2PC) [18]

- Idea: one process coordinates commit and checks that all agree on decision

Sketch of the algorithm

1. Prepare phase
    1. Coordinator sends message with transaction to all participants
    2. Participant executes transaction until commit is needed.
       Replies yes (commit) or no (e.g. conflict). Records changes in undo/redo logs
    3. Coordinator checks decision by all replies, if all reply yes, decide commit
2. Commit phase
    1. Coordinator sends message to all processes with decision
    2. Processes commit or rollback the transactions, send acknowledgment
    3. Coordinator sends reply to requester

- Think about: What should happen if the coordinator fails?
- What should a "participant" do upon such failures, how to detect them?

2PC for our Bank Example [21]

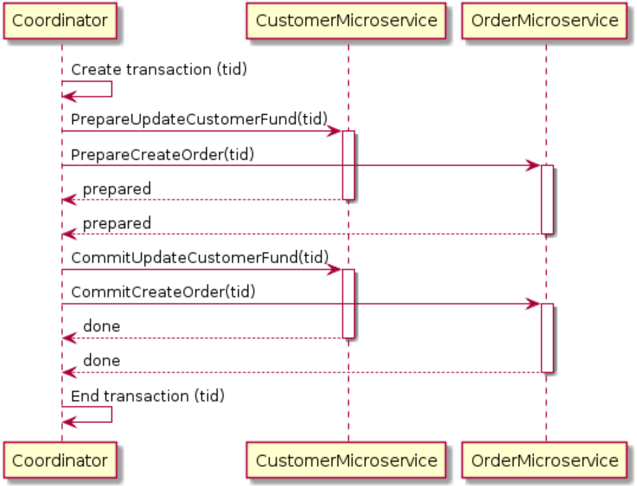


Figure: Source: [21]

Consistent Hashing

- Goal: manage key/value data in a distributed system
  - Load balancing, i.e., all nodes have similar amount of keys
  - Faul tolerant, deal with loss of nodes / adding of nodes
- Idea: distribute keys and servers (capabilities) on a ring (0-(M-1))
  - Fault tolerance: store item multiple times by hashing key multiple times
    - Multiple hash functions could be used
  - Load balancing: hash server multiple times on the ring (e.g., 10x)
- Data allocation: the server with the next bigger number is responsible
- Upon server failure, the items on the server must be replicated again
- See `https://www.youtube.com/watch?v=juxlRh4ZhoI` and [22], [23]

Consistent Hashing (2)

- In this example, the IP addresses are hashed to the ring
  - Imagine they are hashed several times (fault tolerance!)
- The items are strings, the hash determines where they are located
- The arrow shows the server responsible for the items

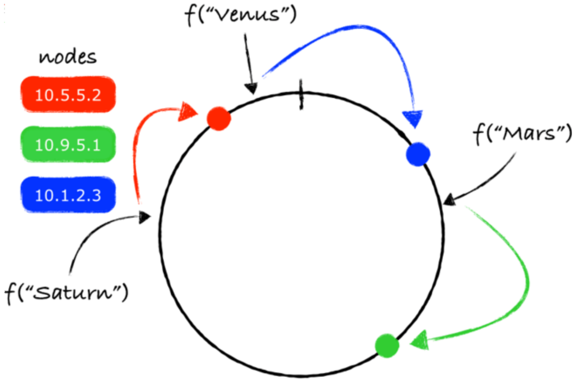


Figure: Source: [22]

Outline

REST Architecture

- Representational state transfer (REST) softare architecture
- **RESTful**: Term indicates the system is conforming to REST constraints

Architectural Constraints

- Client-server architecture
- Statelessness: server do not have to keep any state information
- Cacheability: responses can be cached (answer the same request)
- Layered system: can utilize proxy (intermediate) or load-balancer
- Uniform interface: System API utilizes HTTP/TCP
- Code on demand (optional): deliver code that is executed on the server

REST [31]

- Advantages due to HTTP
  - Simplicity of the interfaces
  - Portability: Independent of client and server platform
  - Cachable: Read requests can be cached close to the user
  - Tracable: Communication can be inspected

Semantics of HTTP request verbs [33]

- GET: retrieve a representation of a resource (no updates)
- PUT: store the enclosed data under the given URI
- POST: transfer an entity/data as subordinate of the web resource
- DELETE: remove the given URI
- PUT and DELETE are idempotent
  - GET also w/o concurrent updates

HTTP 1.1 [33]

- The Hypertext Transfer Protocol (HTTP) is a stateless protocol
- Request via TCP $\Rightarrow$ Response (status and content) via TCP
- Request/Response are encoded in ASCII
- Include a header with standardized key/value pairs [34]
- Non-standard key/value pairs can be added
  - Usually prefixed with X for eXtension
- One data section (at the end) according to the media type
- Separation between header and data via one newline

Example HTTP Request

```
GET /dir/file HTTP/1.1
Host: www.test.de:50070
User-Agent: mozilla
Cache-Control: no-cache
Accept: */*
```

HTTP 1.1

Media types [35]

- Based on Multipurpose Internet Mail Extensions (MIME) types
- Media type is composed of type, subtype and optional parameters
  - e.g., image/png
  - e.g., text/html; charset=UTF-8
- Media types should be registered by the IANA[3]

Example HTTP Response

```
HTTP/1.1 200 OK
Date: Sun, 06 Dec 2015 16:41:16 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: PREF=ID=11111:FF=0:TM=1449420076:LM=1444476:V=1:S=doDl; expires=Thu, 31-Dec-2015 16:02:17 GMT; path=/; domain=.test.de
Set-Cookie: NID=74=UNTSNZy expires=Mon, 06-Jun-2016 16:41:16 GMT; path=/dir; domain=.test.de; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked

DATA formatted according to content type
```

REST Semantics

- Depends on the service implementation
- Behavior usually depends on URI type
  - Collections/Directories, e.g., `http://test.de/col/`
  - Items/Files, e.g., `http://test.de/col/file`

Typical semantics [31]

Resource	GET	PUT	POST	DELETE
Collection	List the collection	Replace the collection with new data.	Create a new entry in the collection, return the URI of the created entry	Delete the collection
Item	Retrieve the data	Replace the element or create it	Not widely used.	Delete the element in the collection

- Provide compatible semantics as responses may be cached!
- POST is highly flexible

Direct API Access via TCP

- Connect to the service IP address and port via TCP
- Use any API or tool, for example:
  - UNIX sockets for C, Python, ...
  - Netcat (nc)
  - `curl`
  - Python
  - Browser

CURL

- curl transfers data from/to a server
- Useful for scripting / testing of webservers
- Supports many protocols, standards for proxy, authentication, cookies, ...

```
# -i: include the HTTP header in the output for better debugging
# -L: if the target location has moved, redo the request on the new location
curl -i -L "http://xy/bla"
# Send data provided in myFile using HTTP PUT, use "-" to read from STDIN
curl -i --request PUT "http://xy/bla?param=x&y=z" -d "@myFile"
# To put a binary file use --data-binary
curl -i --request POST --data-binary "@myFile" "http://xy/bla?param=x&y=z"
# Delete a URI
curl -i -request DELETE "http://xy/bla?param=x&y=z"
```

Python

- The requests package supports HTTP requests quite well

Transferring JSON data

```
import json, requests

params = {'parameters' : [ 'testWorld' ] }

s = requests.Session() # we use a session in this example
resp = s.post(url     = 'http://localhost:5000/compile',
              data    = json.dumps(params),
              headers = {'content-type': 'application/json'},
              auth    = ('testuser','my secret'))
print(resp.status_code)
print(resp.headers)

# assume the response is in JSON
data = json.loads(resp.text, encoding="utf-8")

# retrieve another URL using HTTP GET
resp = s.get(url='http://localhost:5000/status', auth=('testuser','my secret'))
```

Example: WebHDFS [32]

- Full access to file system via `http://$host/webhdfs/v1/FILENAME?op=OPERATION`

```
$ host=10.0.0.61:50070
$ curl -i -L "http://$host/webhdfs/v1/foo/bar?op=OPEN"
HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Sun, 06 Dec 2015 16:06:11 GMT
Date: Sun, 06 Dec 2015 16:06:11 GMT
Pragma: no-cache
Content-Type: application/octet-stream
Location: http://abu1.cluster:50075/webhdfs/v1/foo/bar/file?op=OPEN&namenoderpcaddress=abu1.cluster:8020&offset=0
Content-Length: 0
Server: Jetty(6.1.26.hwx)

HTTP/1.1 200 OK
Access-Control-Allow-Methods: GET
Access-Control-Allow-Origin: *
Content-Type: application/octet-stream
Connection: close
Content-Length: 925
DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA DATA

$ curl -i "http://$host/webhdfs/v1/?op=GETFILESTATUS"
HTTP/1.1 200 OK
Cache-Control: no-cache
Expires: Sun, 06 Dec 2015 16:11:14 GMT
Date: Sun, 06 Dec 2015 16:11:14 GMT
Pragma: no-cache
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(6.1.26.hwx)
{"FileStatus":{"accessTime":0,"blockSize":0,"childrenNum":7,"fileId":16385,"group":"hdfs","length":0,"modificationTime":
1444759104314,"owner":"hdfs","pathSuffix":"","permission":"755","replication":0,"storagePolicy":0,"type":"DIRECTORY"}}
```

Summary

- Designing a distributed system/algorithm requires to think about
  - required functionality
  - semantics (API, properties)
  - Important properties: Availability, Consistency, Fault-tolerance
- Architectural-patterns provide blueprints for distributed systems
- The CAP-theorem describes the limitations of ACID systems
  - It lead to the BASE semantics of NoSQL-solutions
- BigData systems benefit from the Lambda architecture
- The REST architecture is build on top of HTTP and portable
  - Caching of HTTP is important to increase scalability!

Bibliography

4 Forrester Big Data Webinar. Holger Kisker, Martha Bennet. Big Data: Gold Rush Or Illusion?

10 Wikipedia

11 Book: N. Marz, J. Warren. Big Data – Principles and best practices of scalable real-time data systems.

12 https://en.wikipedia.org/wiki/Data_model

14 https://en.wikipedia.org/wiki/Programming_paradigm

15 https://en.wiktionary.org/wiki/process

16 https://en.wikipedia.org/wiki/Paxos_(computer_science)

17 https://en.wikipedia.org/wiki/Consensus_(computer_science)

18 https://en.wikipedia.org/wiki/Two-phase_commit_protocol

19 https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns#Distributed_systems

20 https://en.wikipedia.org/wiki/Distributed_algorithm

21 https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/

22 https://akshatm.svbtle.com/consistent-hash-rings-theory-and-implementation

23 https://www.toptal.com/big-data/consistent-hashing

24 https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture

26 Overcoming CAP with Consistent Soft-State Replication https://www.cs.cornell.edu/Projects/mrc/IEEE-CAP.16.pdf

27 https://en.wikipedia.org/wiki/Multitier_architecture

31 https://en.wikipedia.org/wiki/Representational_state_transfer

32 http://hortonworks.com/blog/webhdfs-%E2%80%93-http-rest-access-to-hdfs/

33 https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

34 https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

35 https://en.wikipedia.org/wiki/Media_type