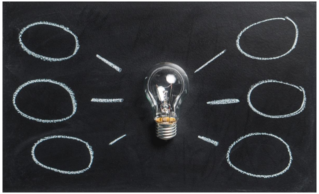
GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Department of Computer Science

Julian Kunkel

Designing Distributed Systems and Performance Modelling



Learning Objectives

- List example problems for distributed systems
- Sketch the algorithms for two-phase commit and consistent hashing
- Discuss semantics and limitations when designing distributed systems
- Explain the meaning of the CAP-theorem
- Sketch the 3-tier architecture
- Design systems using the RESTful architecture
- Describing relevant performance factors for HPDA
- Listing peak performance of relevant components
- Assessing/Judging observed application performance

Outline

1. Motivation Example: Big Data
2. Distributed Algorithms
3. Example Problems
4. REST Architecture
5. High-Level Performance
6. System Characteristics
7. Assessing Performance
8. Summary

Components for High-Performance Data Analytics

Required components

- 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[35]
- Data Warehouses with schemas are useful for well known repeated analysis

Processing capabilities

- Performance is important; goal: interactive processing is difficult
- Available technology offers
  - Batch processing (hours to a day processing time)
  - "Real-time" processing (seconds to minutes turnaround)

[35] See `http://nosql-database.org/` for a big list.

Basic Considerations for High-Performance Analytics

Analysis requires efficient (real-time) processing of data

- New data is continuously 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 can we program data processing systems and services? - Distributed algorithms
  - 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)?
  - How can assess observed performance, i.e., what performance can we expect?

Outline

1 Motivation Example: Big Data

2 Distributed Algorithms

3 Example Problems

4 REST Architecture

5 High-Level Performance

6 System Characteristics

7 Assessing Performance

8 Summary

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 of a Service

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

- Application programming interface (API)
- **Consistency**: Behaviour 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 behaviour depending on the number of clients, concurrent requests, request size, etc.
  - Partition tolerance: Continue to operate even if the 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., still available during interruptions
- Fault-tolerance, i.e., sensible error handling
- Scalable, i.e., handles increased load well
  - Linear scalability with the data volume (or number of users served)
    - 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, i.e., simple programming models
- Ready for the cloud, i.e., can be packaged and deployed as containers
- Debuggability, i.e., errors and events can be logged and traced
  - In respect to coding errors and performance issues
- High Performance
  - Real-time/interactive capabilities - user interact with the system without noticing delay
- High efficiency, i.e., make good use of resources (compute and storage)

Consistency Limitations in Distributed Systems

- Communication is essential in a distributed system but faults are common
- Partitioning: what happens if one half of the system can't communicate with the other?

CAP-Theorem

- It initially discusses implications, if the network is partitioned[36]
  - 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

⇒ 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/

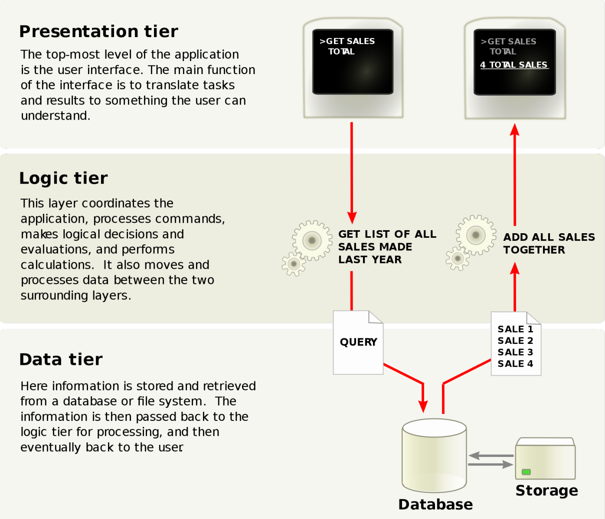
[36] This means that network failures split the network peers into multiple clusters that cannot communicate.

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 on any compute node
- 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.

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.

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.

Figure: Source: [25]

Object Request Broker [24]

- Example: Common Object Request Broker Architecture (CORBA)
  - Example of the distributed object paradigm: objects 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

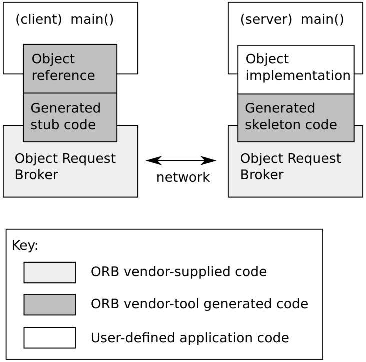


Figure: Example code. Source: Alksentrs, [24]

Object Broker: Code Example

Client Interface

```
public User(){
  public:
  // load user details from given UserID
  User(string userID);

  // allow users to change the username to a new name
  int changeUserName(string username);

  // typical functions to get some data
  string getName();
};
```

Client Stub Code

```
class RemoteUser(public User){
  private:
    Server server; // responsible for this object
    ID       remoteObjectID;
  public:
  RemoteUser(string userID) : User(string userID){
    server = // somehow identify a remote server
    // create the remote object on the server loading the data
    Arguments args;
    args.addStringArgument("userID", userID);
    ID = server.RMI("createRemoteUser", args);
  }

  int changeUserName(string username){
    Arguments args;
    args.appendStringArgument("username", username);
    // handle server faults
    try{
      Message result = server.RMI(ID, "changeUserName", args);
    }catch(...){
      // could try to load user data on another server
      // assign a new server and object ID etc...
    }
    return result;
  }
};
```

- Note that such code would be automatically generated!

Outline

1 Motivation Example: Big Data

2 Distributed Algorithms

3 Example Problems

4 REST Architecture

5 High-Level Performance

6 System Characteristics

7 Assessing Performance

8 Summary

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] (Simplified Consensus Algorithm)

- 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

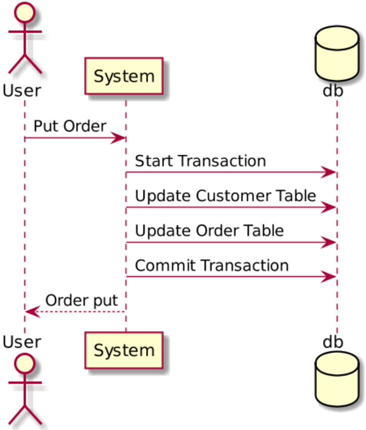


Figure: Execution UML; Source: [21]

Microservice Architecture for Bank Example [21]

- An architecture for tables split across nodes (e.g., via microservices)

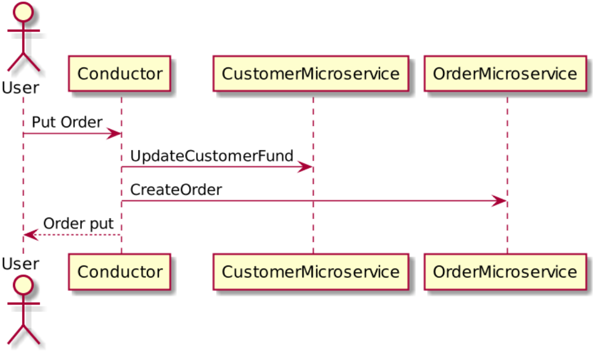


Figure: Source: [21]

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

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

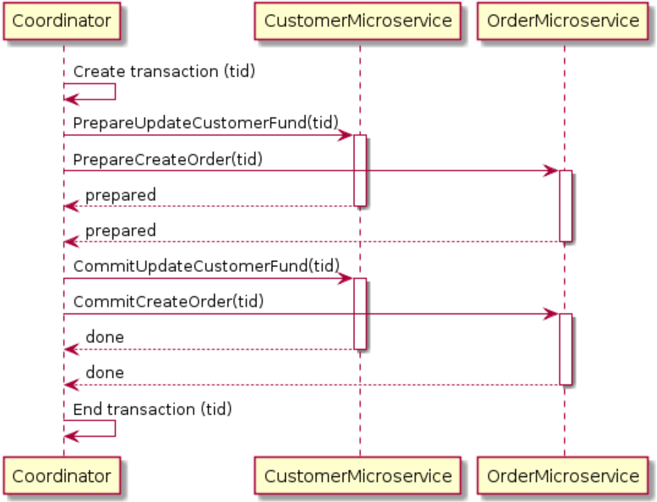


Figure: Source: [21]

Consistent Hashing

- Goal: manage key/value data in a distributed system
  - Load balancing, i.e., all nodes have a similar number of keys
  - Fault tolerant, deal with the 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
    - Different hash functions could be used or multiple hashing rounds
  - 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
- Adding/removing servers will only transfer subset of the data

Consistent Hashing (2)

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

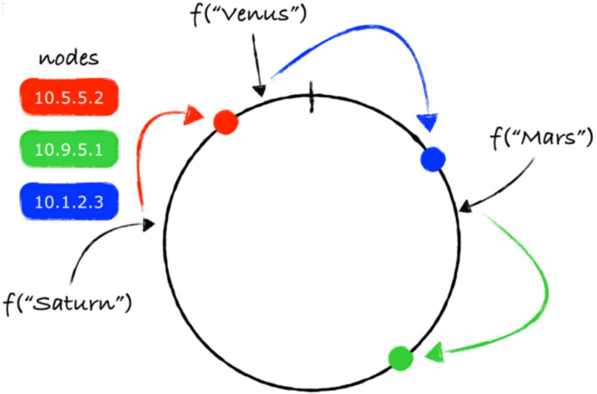


Figure: Source: [22]

- For more info, see `https://www.youtube.com/watch?v=juxlRh4ZhoI` and [22], [23]

Outline

1 Motivation Example: Big Data

2 Distributed Algorithms

3 Example Problems

4 REST Architecture

5 High-Level Performance

6 System Characteristics

7 Assessing Performance

8 Summary

REST Architecture

- Representational state transfer (REST) software architecture
  - REST APIs are the backbone of various distributed applications
  - **RESTful**: Term indicates the system is conforming to REST constraints
  - REST is an architectural style and NOT a protocol

Architectural Constraints / Features

- Client-server architecture
- Statelessness: server does not have to keep any state information
- Cacheability: responses can be cached when labelled so
- Layered system: can utilize proxy (intermediate) or load-balancer
- Uniform interface: Self-descriptive hypermedia messages
  - Originally: Responses must be in HTML and descriptive (no docs required)
  - Nowadays: wide interpretation of other outputs such as JSON (useful in Webpages)
- Code on demand (optional): deliver code to extend client functionality

REST via HTTP [31]

By definition REST is not bound to using HTTP

- Advantages of REST due to HTTP
  - URI handled via URL path in request
  - 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 methods [33]

- GET: retrieve a representation of a resource (no updates)
- PUT: create or update resource based on enclosed data
- POST: transfer enclosed data to be processed by server
- DELETE: remove the given URI
- PUT and DELETE are idempotent
  - GET also w/o concurrent updates

HTTP Methods

- Depends on the service implementation
- Behaviour usually depends on URI type
  - Collections/Directories, e.g., `http://test.de/col/`
  - Items/Files, e.g., `http://test.de/col/file`
- Define status codes (e.g., 200 OK, 404 Not Found)

Most common semantics [33]

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

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

HTTP 1.1 [33]

- The Hypertext Transfer Protocol (HTTP) is a stateless protocol
- Request via TCP ⇒ 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 Multi-purpose 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[37]

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
```

[37] Internet Assigned Numbers Authority

HTTP 2.0+ [50]

- HTTP 2 is a semantically compatible update of HTTP 1.1 for performance
  - Data compression of HTTP headers
  - HTTP/2 Server Push
  - Pipelining of requests
  - Multiplexing multiple requests over a single TCP connection

HTTP 3.0

- In 2022, still an Internet draft
- Utilizes QUIC (UDP-based) transport layer network protocol instead of TCP
- Fixes head-of-line blocking (due to TCP)

Programming: 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, the Hadoop File System [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"}}
```

Outline

1 Motivation Example: Big Data

2 Distributed Algorithms

3 Example Problems

4 REST Architecture

5 High-Level Performance

6 System Characteristics

7 Assessing Performance

8 Summary

Goals

- In the context of this lecture, we assume the **goal of a system** is data processing
- Goal (user perspective): Minimal time to solution
  - For Big Data: Workflow from data ingestion, programming, results analysis
  - For Science: Workflow until scientific insight/paper
  - Programmer/User productivity is important
- Goal (system perspective): cheap total cost of ownership
  - Simple deployment and easy management
  - Cheap hardware
  - Good utilisation of (hardware) resources means less hardware

⇒ In this lecture, we focus on **processing a workflow**

- Other "performance" alike aspects:
  - Productivity of users (user-friendliness)
  - Energy-efficiency
  - Cost-efficiency

Processing Steps

1. Preparing input data
   - Big Data: Ingesting data into our big data environment
   - HPC: Preparing data for being read on a supercomputer
2. **Processing** a workflow consisting of multiple steps/queries
   - It is a relevant factor for the productivity in data science
   - Low runtime is crucial for repeated analysis and interactive exploration
   - Multiple steps/different tools can be involved in a complex workflow

     For our model, we consider only the execution of one job with any tool
3. Post-processing of output with (external) tools to produce insight
   - Typical strategy of scientists: HPC/Big Data workflow – data transfer – local analysis
   - Best: return a final product from the workflow
   - For exploratory/novel research, the result is unknown, and may require a long period of manual analysis

Performance Factors Influencing Processing Time

- Startup phase
  - Distribution of necessary files/scripts
  - Allocating resources/containers
  - Starting the scripts and loading dependencies
  - Usually fixed costs (in the order of seconds to spawn MR/TEZ job, also for HPC jobs!)
- **Job execution**: computing the product
  - Costs for computation and necessary communication and I/O depending on
    - Job complexity
    - Software architecture of the big data solution
    - Hardware performance and cluster architecture
- Cleanup phase
  - Teardown compute environment, free resources
  - Usually fixed costs (in the order of seconds)

Outline

1 Motivation Example: Big Data

2 Distributed Algorithms

3 Example Problems

4 REST Architecture

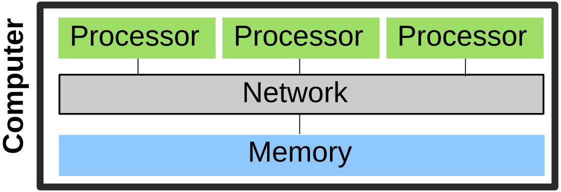
5 High-Level Performance

6 System Characteristics
- Big Data Clusters
- HPC Clusters
- Software

Reminder: Parallel & Distributed Architectures

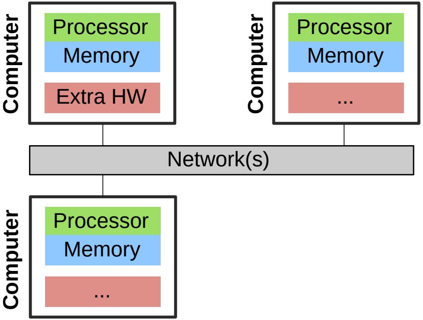
In practice, systems are a mix of two paradigms:

Shared memory



- Processors can access a joint memory
  - Enables communication/coordination
- Cannot be scaled up to any size
- Very expensive to build one big system

Distributed memory systems



- Processor can only see own memory
- Performance of the network is key

Big Data Cluster Characteristics

- Usually commodity components
- Cheap (on-board) interconnect, node-local storage
- Communication (bisection) bandwidth between different racks is low

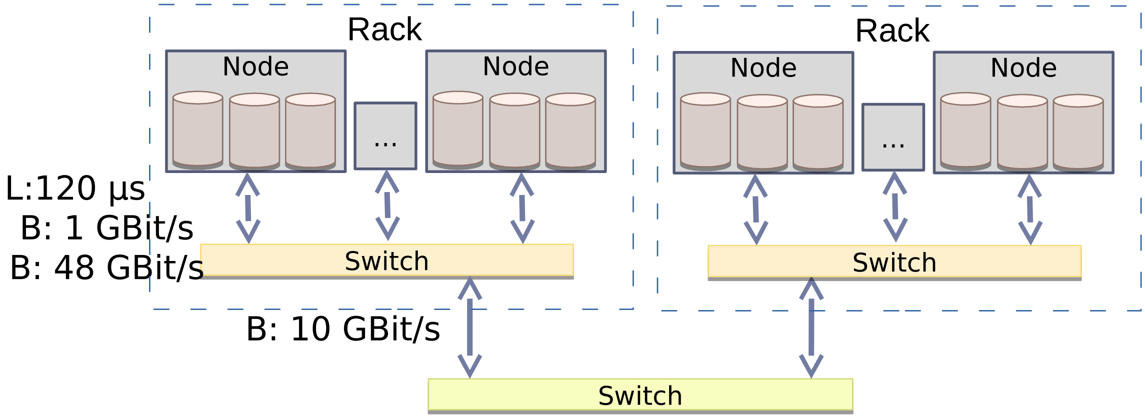


Figure: Architecture of a typical big data cluster

HPC Cluster Characteristics

- High-end components
- Extra fast interconnect, global/shared storage with dedicated servers
- Network provides high (near-full) bisection bandwidth. Various topologies are possible.

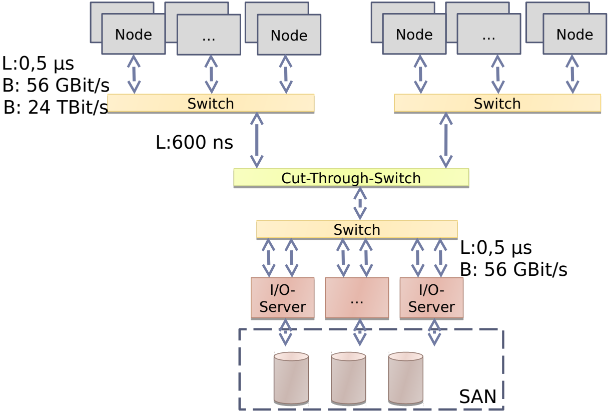


Figure: Architecture of a typical HPC cluster (here fat-tree network topology)

Hardware Performance

Computation

- CPU performance (frequency × cores × sockets)
  - E.g.: 2.5 GHz × 12 cores × 2 sockets = 60 Gcycles/s
  - The number of cycles per operation depend on the instruction stream
- Memory (throughput × channels)
  - E.g.: 25.6 GB/s per DDR4 DIMM × 3

Communication via the network

- Throughput, e.g., 125 MiB/s with Gigabit Ethernet
- Latency, e.g., 0.1 ms with Gigabit Ethernet

Input/output devices

- HDD mechanical parts (head, rotation) lead to expensive seek

⇒ Access data consecutively and not randomly

⇒ Performance depends on the I/O granularity

- E.g.: 150 MiB/s with 10 MiB blocks

Hardware-Aware Strategies for Software Solutions

- Java is suboptimal: 1.2x - 2x of cycles compared to C[38]
- Utilise different hardware components concurrently
  - Pipeline computation, I/O, and communication
  - At best hide two of them $\Rightarrow$ 3x speedup vs sequential
  - Avoid barriers (waiting for the slowest component)
- Balance and distribute workload among all available servers
  - Linear scalability is vital (and not the programming language)
  - Add 10x servers, achieve 10x performance (or process 10x data)
- Allow monitoring of components to see their utilisation
- Avoid I/O, if possible (keep data in memory)
- Avoid communication, if possible

Examples for exploiting locality in SQL/data-flow languages

- Foreach, filter are node-local operations
- Sort, group, join need communication

[38] This does not matter much compared to the other factors. But vectorization matters.

Outline

1 Motivation Example: Big Data

2 Distributed Algorithms

3 Example Problems

4 REST Architecture

5 High-Level Performance

6 System Characteristics

7 Assessing Performance
- Approach
- Assessing Compute and Storage Workflow

Basic Approach

Question

Is the observed performance acceptable?

Basic Approach

Start with a simple model

1. Measure time for the execution of your workload
2. Quantify the workload with some metrics
   - E.g., amount of tuples or data processed, computational operations needed
   - E.g., you may use the statistics output for each Hadoop job
3. Compute W, the workload you process per time
4. Compute the expected performance P based on the system's hardware characteristics
5. Compare W with P, the efficiency is $E = \frac{W}{P}$
   - If $E << 1$, e.g., 0.01, you are using only 1% of the potential!

Refine the model as needed, e.g., include details about intermediate steps

Groupwork: Assessing Performance (Compute Only)

Task: Aggregating 10 Million integers with 1 thread

- Vendor-reported performance from [14] indicates improvements

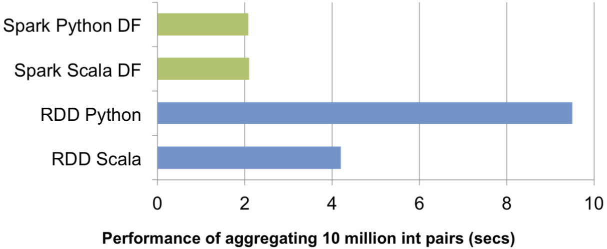


Figure: Source: Reference [14]

- These are the advancements when using Spark for the computation
- Can we trust in such numbers? Are these numbers good?
- Discuss these numbers with your neighbour (Time: 3 minutes)

Assessing Performance of In-Memory Computing

Measured performance numbers and theoretic considerations

- Spark [14]: 160 MB/s, 500 cycles per operation[39]
  - Invoking external programming languages is even more expensive!
- Python (raw): 0.44s = 727 MB/s, 123 cycles per operation
- Numpy: 0.014s = 22.8 GB/s, 4 cycles per operation (memory BW limit)
- One line to measure the performance in Python using Numpy:

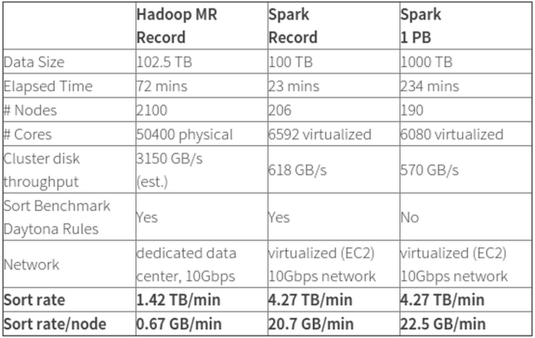
```
timeit.timeit(stmt="np.sum(d)", setup="import numpy as np; d =
    ↪ np.array(range(1,10*1000*1000))", number=1)
```

- Hence, the big data solution is 125x slower in this example than expected!

[39] But it can use multiple threads easily.

Assessing Compute and Storage Workflow

- Daytona GraySort: Sort at least 100 TB data in files into an output file
  - Generates 500 TB of disk I/O and 200 TB of network I/O [12]
  - Drawback: Benchmark is not very compute intense
- Data record: 10 byte key, 90 byte data
- Performance Metric: Sort rate (TBs/minute)



	Hadoop MR Record	**Spark Record**	**Spark 1 PB**
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Figure: Source: Reference [12]

Assessing Performance of In-Memory Computing

Hadoop

- 102.5 TB in 4,328 seconds [13]
- Hardware: 2100 nodes, dual 2.3 Ghz 6cores, 64 GB memory, 12 HDDs
- Sort rate: 23.6 GB/s = 11 MB/s per Node ⇒ 1 MB/s per HDD
- Clearly this is suboptimal!

Apache Spark (on disk)

- 100 TB in 1,406 seconds [13]
- Hardware: 207 Amazon EC2, 2.5 Ghz 32vCores, 244GB memory, 8 SSDs
- Sort rate: 71 GB/s = 344 MB/s per node
- Performance assessment
  - Network: 200 TB ⇒ 687 MiB/s per node
    Optimal: 1.15 GB/s per Node, but we cannot hide (all) communication
  - I/O: 500 TB ⇒ 1.7 GB/s per node = 212 MB/s per SSD
  - Compute: 17 M records/s per node = 0.5 M/s per core = 4700 cycles/record

Executing the Optimal Algorithm on Given Hardware

An utopic algorithm

Assume 200 nodes and well known key distribution

1. Read input file once: 100 TB
2. Pipeline reading and start immediately to scatter data (key): 100 TB
3. Receiving node stores data in likely memory region: 500 GB/node
   Assume this can be pipelined with the receiver
4. Output data to local files: 100 TB

Estimating optimal runtime

Per node: 500 GByte of data; I/O: keep 1.7 GB/s per node

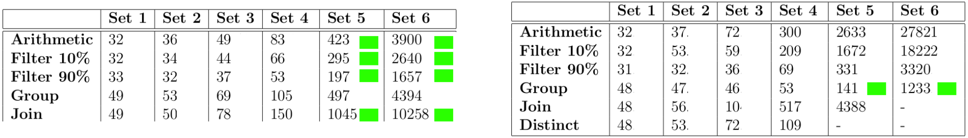
1. Read: 294s
2. Scatter data: 434s $\Rightarrow$ Reading can be hidden
3. One read/write in memory (2 sockets, 3 channels): 6s
4. Write local file region: 294s

Total runtime: $434 + 294 = 728 \Rightarrow$ 8.2 T/min $\Rightarrow$ The Spark record is quite good!

Discussion: Comparing Pig and Hive Big Data Solutions

Benchmark by IBM [16], similar to Apache Benchmark

- Tests several operations, data set increases 10x in size
  - Set 1: 772 KB; 2: 6.4 MB; 3: 63 MB; 4: 628 MB; 5: 6.2 GB; 6: 62 GB
- Five data/compute nodes, configured to run eight reduce and 11 map tasks



	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Arithmetic	32	36	49	83	423 🟩	3900 🟩
Filter 10%	32	34	44	66	295 🟩	2640 🟩
Filter 90%	33	32	37	53	197 🟩	1657 🟩
Group	49	53	69	105	497	4394
Join	49	50	78	150	1045 🟩	10258 🟩

	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Arithmetic	32	37	72	300	2633	27821
Filter 10%	32	53	59	209	1672	18222
Filter 90%	31	32	36	69	331	3320
Group	48	47	46	53	141 🟩	1233 🟩
Join	48	56	10	517	4388	-
Distinct	48	53	72	109	-	-

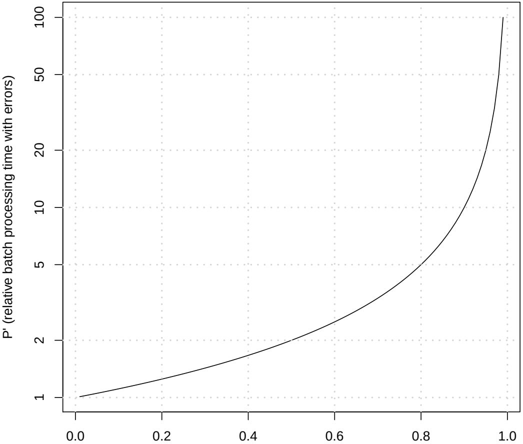
Figure: Time for **Pig (left) and Hive**. Source: B. Jakobus (modified), "Table 2: Averaged performance" [16]

Assessing performance

- How could we model performance here?
- How would you judge the runtime here?

Errors Increase Processing Time [11]

- An error probability $E < 1$ increases the processing time P
- Rerun of a job may fail again
- Processing time with errors can be computed: $\hat{P} = (E + E^2 + ...) \times P = P/(1 - E)$



- With 50% chance of errors, 2x processing time
- With 90% chance, 10x

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 REST architecture is build on top of HTTP and portable
  - Caching of HTTP is important to increase scalability!
- Performance
  - Goal (user-perspective): Optimise the time-to-solution
  - Runtime of queries/scripts is the main contributor
  - Understanding a few HW throughputs help to assess the performance
  - Linear scalability of the architecture is the crucial performance factor
  - Basic performance analysis
    1. Estimate the workload
    2. Compute the workload throughput per node
    3. Compare with hardware capabilities
  - Error model predicts runtime if jobs must be restarted
  - Different big data solutions exhibit different performance behaviours

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://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

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

50 https://en.wikipedia.org/wiki/HTTP/2