Unit 6 - Scalability

Network Information System Technologies

Index

Goals

- Knowledge of the existing compromise between consistency, availability and network partition tolerance when distributed services are used.
- Identification of mechanisms for improving the scalability of distributed services.
- Proper management of a scalable service in order to become elastic.
- Knowledge of the contention problem and identification of mechanisms for its avoidance.

Index

1. Introduction

- Last year, in CSD, you learnt several scalability mechanisms:
  - Process distribution
  - Data distribution
  - Replication
  - Caching
- All they are related with data consistency:
  - Strong consistency demands a high synchronisation among processes
    - This leads to blocking. Therefore, it isn't scalable!
  - Relaxed consistency doesn't require many data exchanges
    - It may be scalable
    - But, sometimes, unexpected results may be generated!

1. Introduction

- There are some aspects of consistency to be considered in scalable services:
  - CAP theorem
    - Impossibility of strong consistency in a geo-replicated service
  - Classical replication models provide strong consistency
    - Other approaches are needed for relaxing consistency!
  - How can the data be distributed among multiple servers?
    - Data partitioning techniques ---> NoSQL datastores

Index

2. CAP Theorem

- In a distributed system these three properties cannot be ensured simultaneously (CAP Theorem):
  - Strong (e.g., sequential) **c**onsistency(C)
  - Service **a**vailability (A)
    - Availability: All clients can access a service
    - Each service replica may have its own clients
      - All clients must be able to get their requests answered in order to consider that the service is available
  - Network-**p**artition tolerance (P)
    - Network-partition: This failure situation implies that at least a system subgroup becomes unable to communicate with all the other subgroups
      - Loss of connectivity
    - Tolerance to partitions implies other aspects are preserved
      - Availability or
      - Consistency
- One of these properties should be sacrificed

2. CAP Theorem

- Why do we need such a theorem??
  - Distributed systems need to ensure failure transparency
  - To this end, provided services should rely on replicated servers
    - This ensures AVAILABILITY (A)
  - The image of a single system should be provided to the user
    - This means that users should be able to observe the same state on all replicas: Sequential CONSISTENCY (C)
  - Failure transparency also implies that the system as a whole does not fail, even in scenarios where connectivity is lost
    - This means that PARTITION TOLERANCE (P) is also needed

2. CAP Theorem

- Option 1: Renouncing to PARTITION TOLERANCE (P)
  - When strong consistency and full service availability need to be maintained, partition tolerance is dropped
    - A partitioned network is not admitted!!
    - What this means??
      - We will try to ensure connectivity
        - For instance, replicating the network
    - But continuous connectivity is hard to ensure
      - This is only affordable in local deployments involving a single lab
      - And even in that case it is extremely difficult to ensure
        - Let us look for other alternatives!!

2. CAP Theorem

- Option 2: Renouncing to AVAILABILITY (A).
  - We want to ensure strong consistency and partition tolerance
  - When a partition occurs, availability suffers
    - Some "primary partition" model can be adopted
      - Every "secondary" subgroup is stopped
        - Clients connecting to them see an unavailable service!!
          - In practice, all those nodes seem to have failed
      - At most ONE single "primary" subgroup may go on
        - All their nodes can maintain an internal strong consistency
        - E.g. the subgroup with a majority of nodes

2. CAP Theorem

- Option 3: Renouncing to CONSISTENCY (C).
  - We want to ensure service availability and partition tolerance
  - When a partition arises:
    - The system admits that all subgroups go on
      - Partitionable model
    - When update operations are processed,
      - Only one subgroup applies them
        - Strong consistency is lost
    - A carefully designed system, with commutative update operations, ensures eventual consistency
      - Trivial reconciliation procedures

2. CAP Theorem

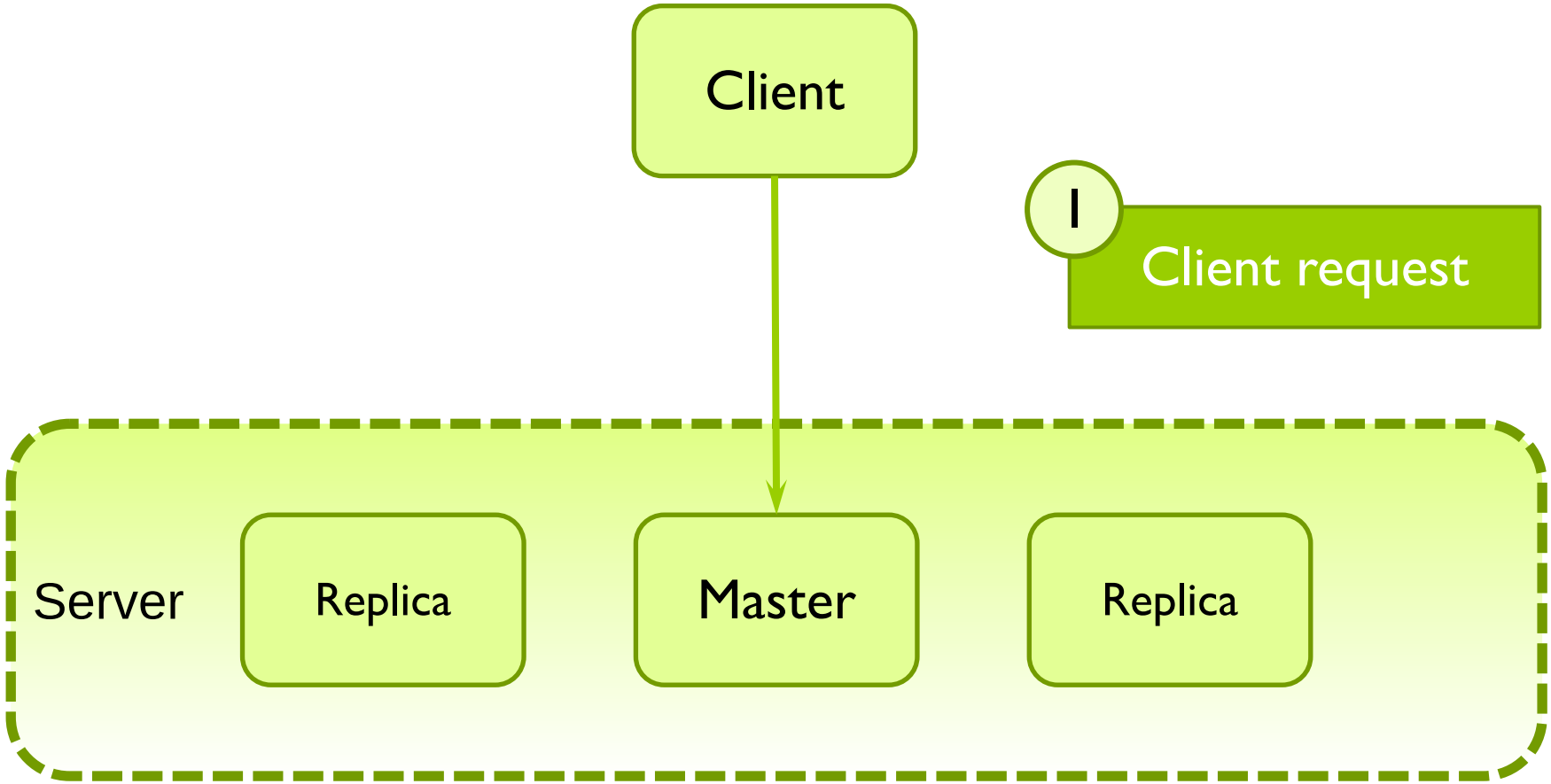
- Which is the best option?
  - Scalable systems should ensure availability
    - Their goal is to serve a large (and potentially increasing) number of users
    - Once deployed in multiple data centres, it will not be rare that some racks become temporarily disconnected
      - This partition situation should be also admitted
    - Therefore, strong consistency is sacrificed
      - Indeed, that sacrifice is not a novelty since it is already a requirement for being highly scalable

Index

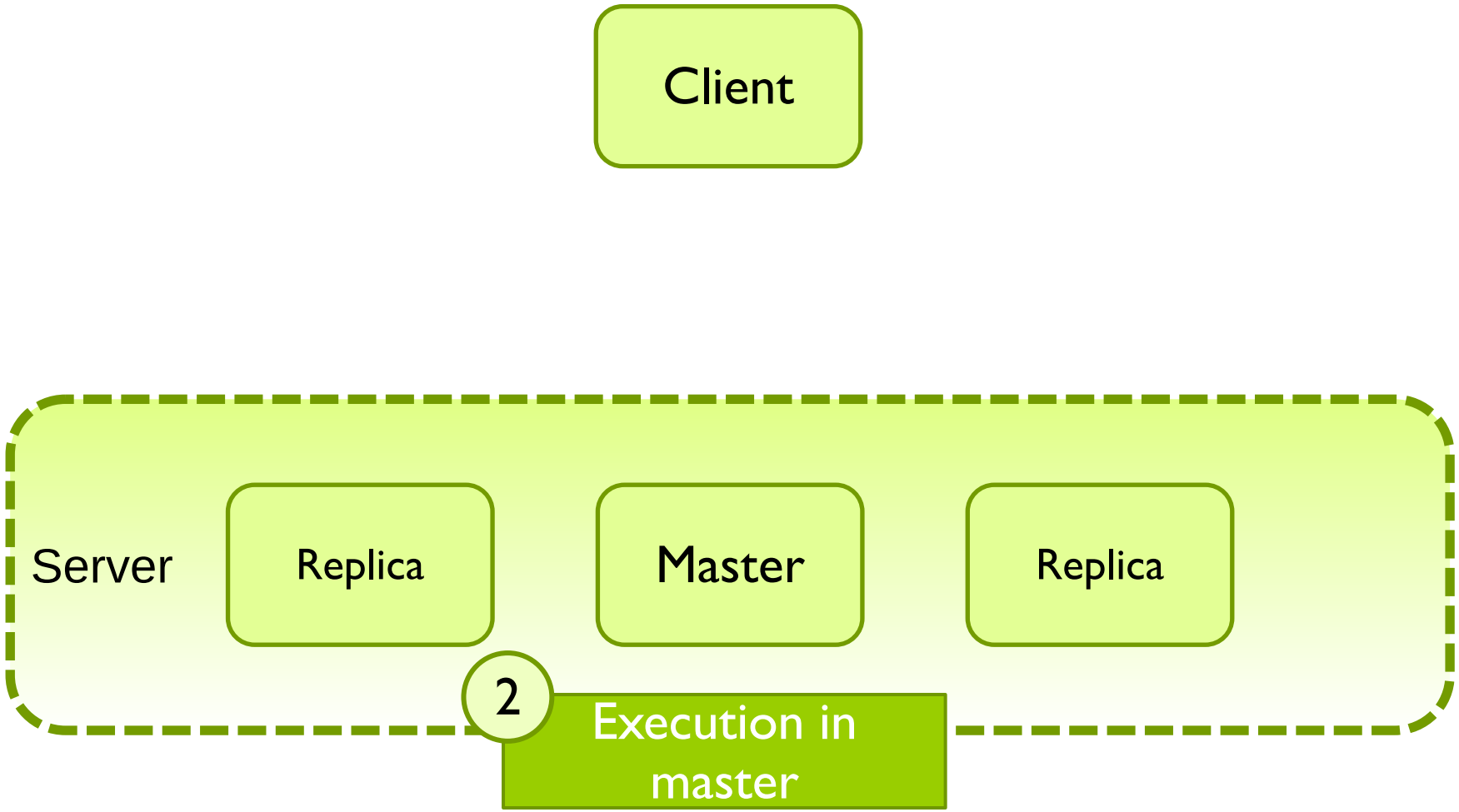
3. Multi-master Replication

- There are two classical replication models:
  - Passive (or "primary/backup replication")
  - Active (or "state-machine replication")
- Unfortunately, both are strongly consistent
  - This may complicate their scalability
- A third solution exists:
  - Multi-master replication
    - Based on the passive model, but...
      - There is no single primary replica
      - Each request may be directly served by a single "master" replica
        - **But each request may use a different master!**
      - The reply is sent immediately to the client
      - Updates are forwarded in a lazy way to the remaining replicas

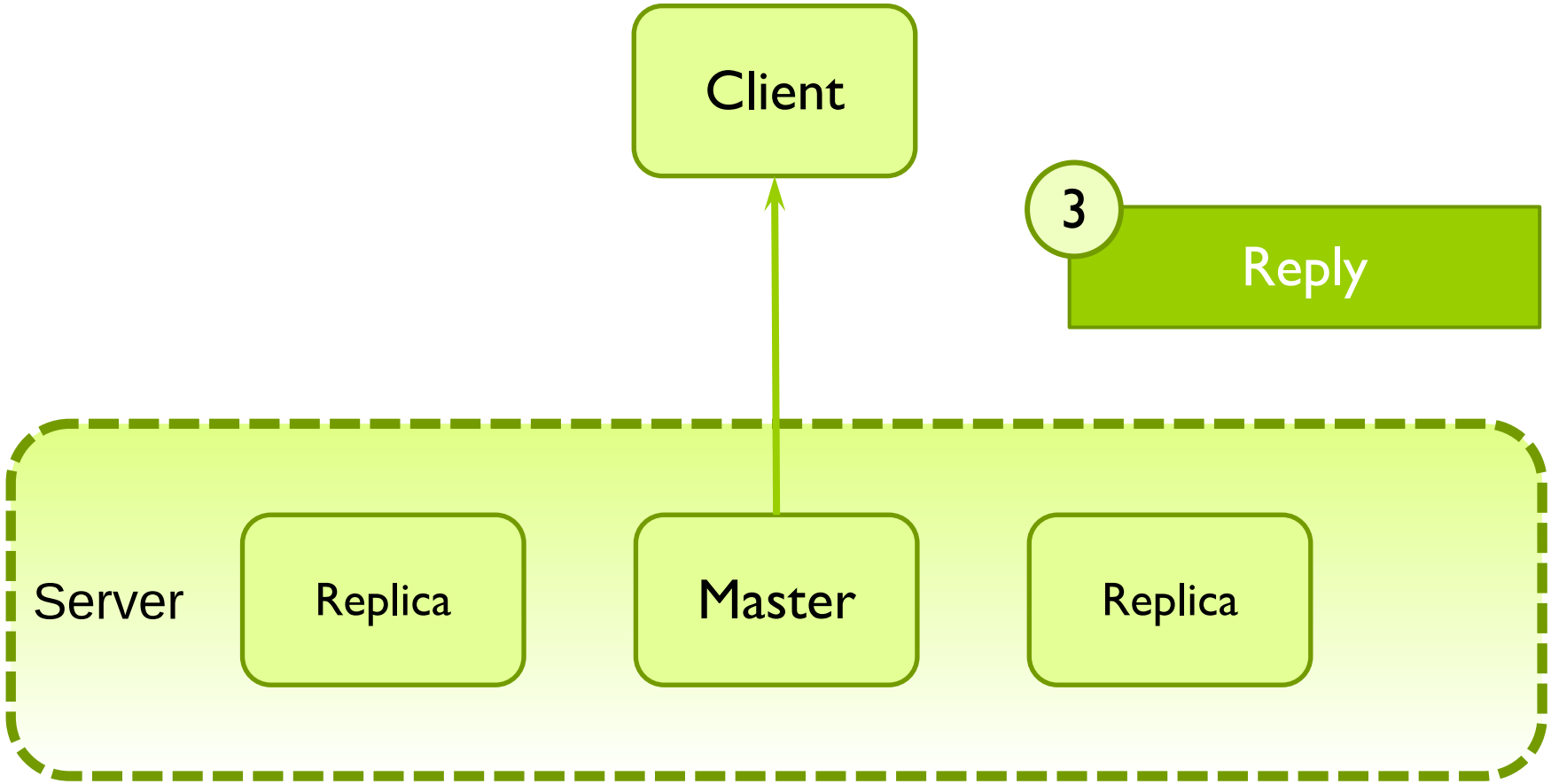
3. Multi-master Replication



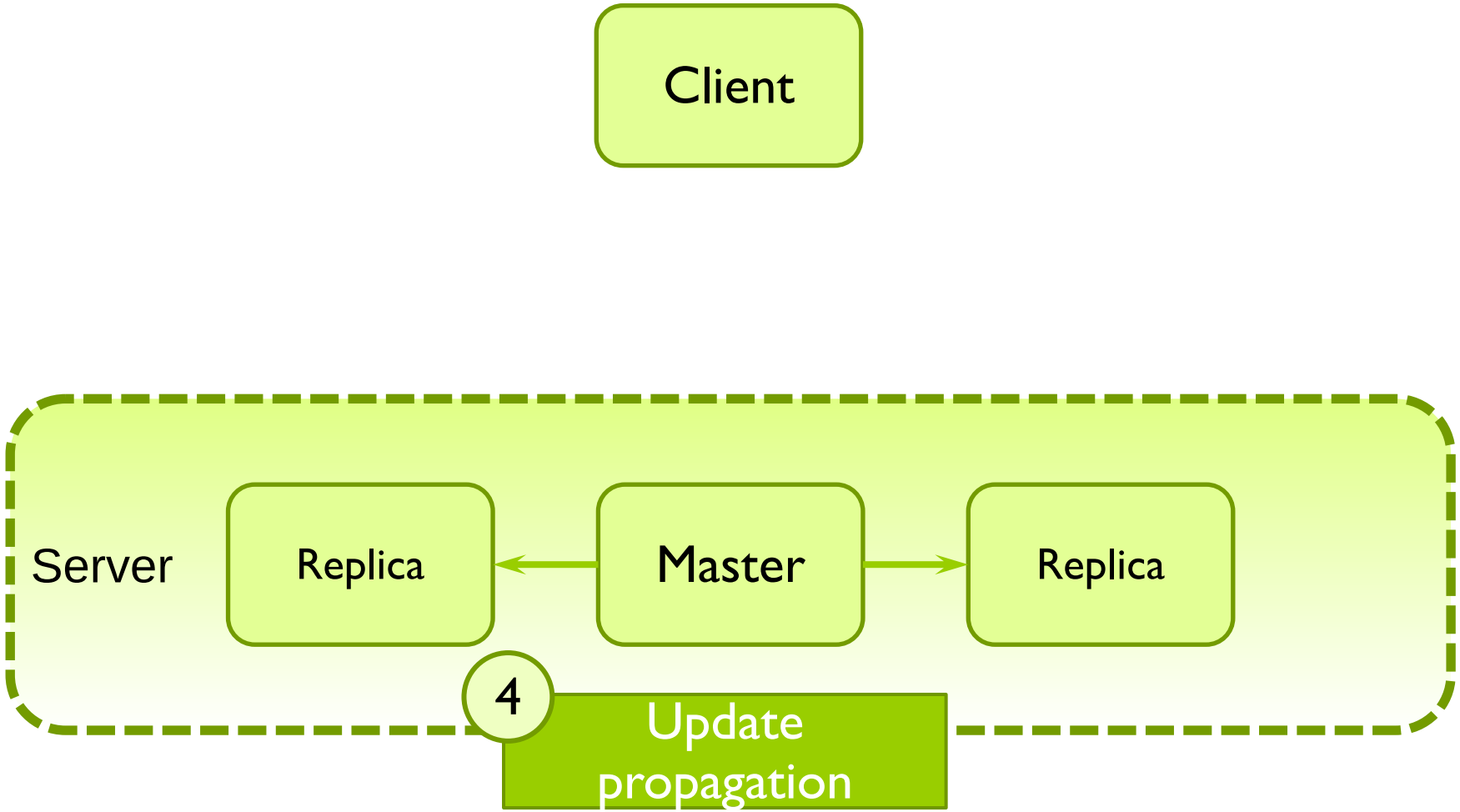
3. Multi-master Replication



3. Multi-master Replication



3. Multi-master Replication



3. Multi-master Replication

- Advantages
  - Minimum overhead
    - Each request is served by a single replica
      - For both read-only and update requests
  - Highly scalable
    - No concurrency control mechanisms are assumed!
  - Admits non-deterministic operations
    - They are only executed by a master
    - Per-operation inconsistencies among replicas never arise

3. Multi-master Replication

- Inconveniences
  - Problems in case of failure
    - When a master fails...
      - Ongoing requests might be lost
  - Inherits the weak failure model management of the passive model
    - Arbitrary failures cannot be managed
  - Inconsistencies may easily arise
    - When concurrent operations served by different masters update the same shared data element
      - This may be handled by the service program if it was written assuming eventual consistency
      - But this is a new responsibility for programmers!

Index

4. NoSQL Stores

- Approaches to improve database scalability:
  - Simplify the schema
    - Replacing a set of (potentially complex) relational tables with simple key-value ones
      - Simple indexes
    - This simplifies also the query language
      - Immediate processing
    - The space needed by the database is also reduced
      - Databases may be kept in main memory
      - Durability ensured by replication
  - Renounce to transactions
    - Transactions will no longer group and protect multiple sentences
      - Atomicity limited to each individual sentence
        - Very short blocking intervals: indiscernible
- Result: NoSQL datastores

4. NoSQL Stores

- NoSQL data-store variants:
  1. Key-value stores
  2. Document stores
  3. Extensible record stores

4.1. Key-Value Stores

- Scheme composed by two attributes: key and value
- Without support for distinguishing sub-attributes in the "value"
- There is no way for querying about non-primary attributes
- Examples: Dynamo, Voldemort, Riak…

4.2. Document Stores

- Schema composed by objects with a variable number of attributes
- In some cases these attributes may be other objects
- Query language based on setting constraints on the attribute values
- Examples: CouchDB, MongoDB, SimpleDB…

4.3. Extensible Record Stores

- Schemas based on tables with a variable number of columns
  - In some cases, they may be organised in column groups
- Tables may be horizontally and vertically partitioned
  - This technique is known as "sharding".
  - Management responsibilities are distributed among multiple nodes
  - Easy load balancing
  - Highly concurrent
  - Easily scalable to multiple servers!
- Examples: Bigtable, PNUTs, Cassandra…

Index

5. Elasticity

- A system is elastic when it is:
  - Scalable
  - Adaptive
    - Each application receives the exact amount of needed resources
    - This adaptability is:
      - Dynamic: It depends on the current workload
      - Autonomous: The system manages its resources without human intervention
- Definition:
  - "Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible".

5. Elasticity

- Elasticity is important in current cloud systems...
  - in order to reduce user costs.
  - in order to manage resource provisioning.
- It requires:
  - A monitoring system for...:
    - current workload.
    - current throughput.
  - A reactive system:
    - For automatizing service reconfiguration
      - Adding nodes and other resources when workload is increased.
      - Releasing resources when workload decreases.
    - Depending on the current "*Service Level Agreement*" (SLA).
      - System reaction should be fast in order to comply with the service level objectives.

Index

6. Contention and Bottlenecks

- Distributed servers consist of multiple components
- A careful design is needed in order to avoid contention
  - Contention arises when a component is overloaded, blocking others (which are still able to manage heavier workloads).
- Contention causes:
  - Centralized algorithms for heavy tasks
    - Centralization (i.e., use of a coordinator) only makes sense for reducing network traffic (in consensus-like steps) in light or infrequent tasks.
  - Use of synchronization mechanisms
    - A resource is shared by multiple tasks, there is a critical section and its protection usually leads to contention.
  - Excessive network traffic
    - An inappropriate resource distribution could lead to an increase in remote communication.

6. Contention and Bottlenecks

- How to avoid contention?
  - In centralised cases:
    - Using a decentralized solution when the task to be accomplished is costly.
  - In the shared-resource case:
    - Access serialization, achieved with asynchronous programming.
      - No concurrency. Trivial management. Fast completion of each request.
      - The resulting system is fast and efficient.
        - Less context-switching overhead.
        - Lower middleware and/or operating system intervention.
  - In case of a wrong distribution of resources:
    - Redistribute or replicate resources.
    - Remote accesses become local accesses.

Index

7. Scalability Examples

1. Introduction
2. CAP Theorem
3. Multi-master Replication
4. NoSQL Stores
5. Elasticity
6. Contention and Bottlenecks
7. Scalability Examples
    1. The "cluster" module in Node.js
    2. Scalability using multiple computers
    3. MongoDB
8. Learning Results
9. References

7.1. The "cluster" module in NodeJS

- In NodeJS, processes only have a unique thread.
- When we need multiple instances of a given service, we cannot create other threads in our server process:
  - We need to create other processes, executing the same code.
  - With multiple processes (replicas of a given component), the workload may be balanced among them.
- The "cluster" module in NodeJS provides:
  - An easy management for pools of workers (Node processes).
  - Load balancing among those workers.

7.1. The "cluster" module in NodeJS

- The `cluster` module…
  - Takes advantage of multiprocessor systems, executing a cluster of Node processes on them.
    - To this end, it balances the load among those processes.
  - Allows creating processes that share the port or ports associated to the service being implemented by that "cluster".
  - Provides a `worker` class that models the cluster workers.
    - They are supervised and created by a "master" process.
  - Provides some events ('`fork`', '`online`', '`listening`', '`disconnect`', '`exit`'…) that may be used for controlling the current state of the worker processes.
  - Uses IPC for master-worker communication.

7.1. The "cluster" module in NodeJS

- Basic scheme of a cluster (or worker pool):

```
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;
var server = ... // the server which runs the workers
var port = ...   // the port where the server binds

if (cluster.isMaster) {  // code of the master one
   // fork: create workers
   for (var i=0; i < numCPUs; i++) cluster.fork();
   // listening death of workers
   cluster.on('exit', function(worker, code, signal) {
      console.log('worker', worker.process.pid, 'died');
   });
} else {  // code of any worker
   server.listen(port); // each worker runs the server
}
```

7.1. The "cluster" module in NodeJS

- Example: Cluster of http servers.

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {  // code of the master one
   for (var i=0; i < numCPUs; i++) cluster.fork();
   cluster.on('exit', function(worker, code, signal) {
      console.log('worker', worker.process.pid, 'died');
   });
} else {  // code of any worker
   http.createServer(function(req, res) {
      res.writeHead(200);
      res.end('hello world\n');
   }).listen(8000);
}
```

7.1. The "cluster" module in NodeJS

- Events of a `Cluster` object:
  - '`fork`': When a new worker is created.
  - '`online`': Generated by a forked worker, telling the master that the worker has started its execution.
  - '`listening`': Generated when a worker calls `listen()`
- These events may be used for supervising the workers, associating timeouts to their starting stage. For instance:

```
var timeouts = [];
function errorMsg() { console.error('Something wrong...');}
cluster.on('fork', function(worker) {
   timeouts[worker.id] = setTimeout(errorMsg, 2000);
});
cluster.on('listening', function(worker, address) {
   clearTimeout(timeouts[worker.id]);
});
```

7.1. The "cluster" module in NodeJS

- Events of a `Cluster` object (ii):
  - '`disconnect`': Generated when the IPC channel from a worker is disconnected (due to the worker termination 'exit', or its death 'kill' or its disconnection 'disconnect')
  - '`exit`': Generated when a worker dies.
- These events may be used for detecting worker inactivity, restarting them in that case using fork(). For instance:

```
cluster.on('exit', function(worker, code, signal) {
   console.log('worker %d died (%s). restarting...',
               worker.process.pid, signal || code);
   cluster.fork();
});
```

7.1. The "cluster" module in NodeJS

- The `cluster.workers` object…
  - Is a hash that stores the set of active workers, indexed by their "id".
  - The 'fork' event is raised when a worker is added to `cluster.workers`.
  - The '`disconnect`' and '`exit`' events remove a worker from the cluster.
- The cluster *master* may send messages to a given worker or to all of them.

7.1. The "cluster" module in NodeJS

- Each **`worker`** object (that may be accessed using "*`cluster.worker`*") consists of:
  - Attributes:
    - **`id`** (unique identifier or key in the *`cluster.workers`* hash)
    - **`process`** (its process)
    - **`exitedAfterDisconnect`**(Boolean: `true` when the worker exits voluntarily; `false` when it has been killed by other processes)
  - Methods, that are invoked using its process attribute:
    - For instance: process.send(…);
    - **`send()`**. Needed for sending messages to the master process.
    - **`disconnect()`**
    - **`kill()`**
  - Events:
    - '**`message`**', '**`online`**', '**`listening`**', '**`disconnect`**', '**`exit`**', '**`error`**'

7.1. The "cluster" module in NodeJS

- Master-worker messages ('message' events) may be used, for instance, for counting the service requests:

```
var cluster = require('cluster');
var http = require('http');
if (cluster.isMaster) {
  var numReqs = 0;
  setInterval(function(){ console.log("numReqs =", numReqs);}, 1000);
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd == 'notifyRequest') numReqs++;
  }
  var numCPUs = require('os').cpus().length;
  for (var i=0; i < numCPUs; i++) cluster.fork();
  for (var i in cluster.workers)
    cluster.workers[i].on('message', messageHandler);
} else {
  http.Server(function(req, res) {
    res.writeHead(200); res.end('hello world\n');
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}
```

7. Scalability Examples

1. Introduction
2. CAP Theorem
3. Multi-master Replication
4. NoSQL Stores
5. Elasticity
6. Contention and Bottlenecks
7. Scalability Examples
    1. The "cluster" module in Node.js
    2. Scalability using multiple computers
    3. MongoDB
8. Learning Results
9. References

7.2. Scalability using multiple computers

- The cluster module may be used for taking advantage of multiprocessors or multiple cores in a given processor.
- However, high workloads may overload a multiprocessor computer. In that case, the service should be deployed onto multiple computers.
- Some solutions are:
  - node-http-proxy
  - HAProxy
  - nginx [engine x]

7.2. Scalability using multiple computers

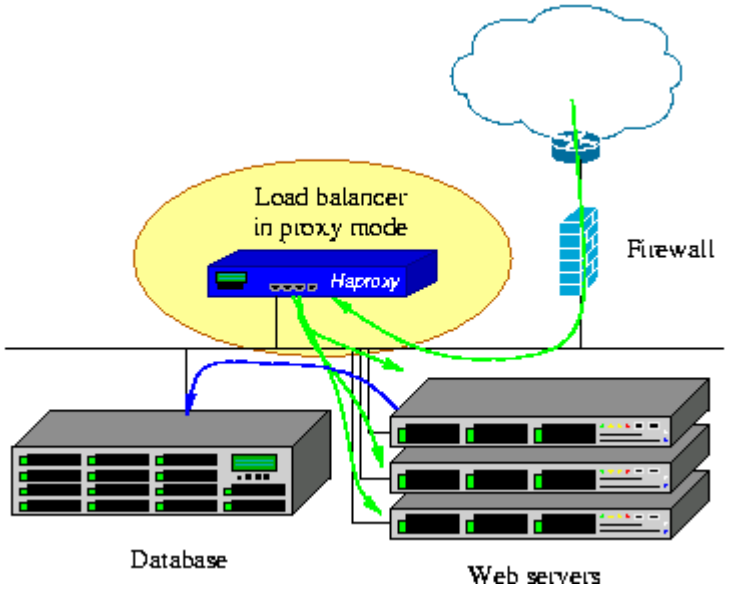
- **node-http-proxy** (https://github.com/nodejitsu/node-http-proxy)
  - It is an open-source proxy server for Node applications.
  - Using it, we may…
    - Configure http server in multiple machines.
    - Balance the workload among all those server instances.
- Basic scheme for this proxy server:

```
var proxyServer = require('http-proxy');
var port = ...
var servers = [{ host: ... , port: ... }, ... ,
               { host: ... , port: ... } ];

proxyServer.createServer(function (req, res, proxy) {
  var target = servers.shift();
  proxy.proxyRequest(req, res, target);
  servers.push(target);
}).listen(port);
```

7.2. Scalability using multiple computers

- **HAProxy** (http://www.haproxy.org/)
  - Open-source proxy server (for TCP and HTTP applications). Written in C.
  - It provides: load balancing, high availability, scalability onto multiple computers.
  - Used in some websites. For instance: GitHub, Stack Overflow, Tumblr, Twitter…



7.2. Scalability using multiple computers

- **HAProxy**. Example: Given this http server in Node.js (*server.js*)…

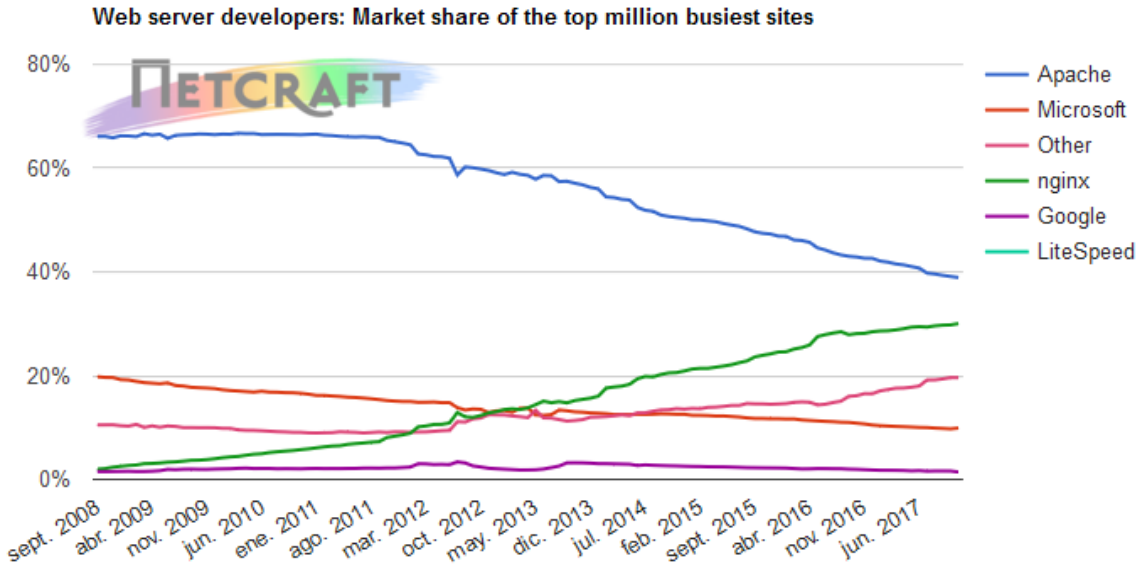
```
var http = require('http');
function serve(ip, port) {
  http.createServer(function (req, res) { ... }
}
serve('0.0.0.0', 9000);
serve('0.0.0.0', 9001);
serve('0.0.0.0', 9002);
```

- …a minimal HAProxy configuration for that service could be…

```
frontend localnodes // where HAProxy listens to connections
  bind *:80
  mode http
  default_backend nodes
backend nodes // where HAProxy sends incoming connections
  mode http
  balance roundrobin
  ... // other configuration options of the backend
  server web01 127.0.0.1:9000 check
  server web02 127.0.0.1:9001 check
  server web03 127.0.0.1:9002 check
```

7.2. Scalability using multiple computers

- **nginx** (http://nginx.org/en/)
  - Open-source reverse proxy server. Developed by Igor Sysoev. Commercially supported by Nginx, Inc.
  - Initially implemented in Russian websites. Currently, it is used world-wide. Indeed, it is used by more than 20% of the most workloaded websites.



7.2. Scalability using multiple computers

- **nginx** (http://nginx.org/en/)
  - Reverse proxy for HTTP and e-mail applications.
  - As all reverse proxies, it may configure multiple HTTP servers in different computers, balancing their workload.
- Basic configuration scheme (load balancing, round robin):

```
http {
  upstream myapp1 {
    server srv1.example.com;
    server srv2.example.com;
    server srv3.example.com;
  }
  server {
    listen 80;
    location / { proxy_pass http://myapp1; }
  }
}
```

7. Scalability Examples

1. Introduction
2. CAP Theorem
3. Multi-master Replication
4. NoSQL Stores
5. Elasticity
6. Contention and Bottlenecks
7. Scalability Examples
    1. The "cluster" module in Node.js
    2. Scalability using multiple computers
    3. MongoDB
8. Learning Results
9. References

7.3. MongoDB

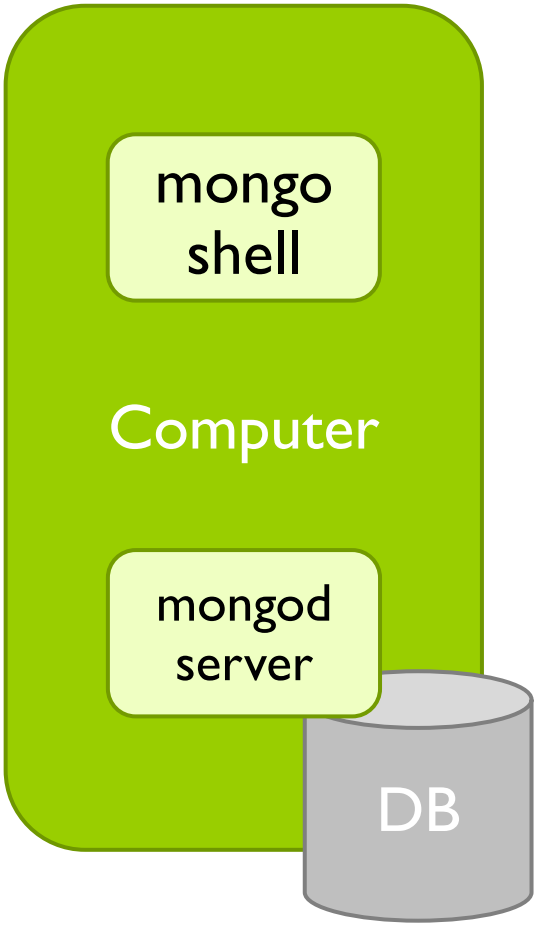
- MongoDB is a scalable "document datastore"
- In a document datastore:
  - The database is a set of collections
    - Collection = Table
  - Each collection is a set of structured objects
    - Each object (or document) has an identifier and multiple attributes
      - The identifier is equivalent to the "primary key" in the relational model
    - Attributes may be structured: sequences of simple types, arrays, objects (with their own attributes),...
      - Every attribute may be seen as a <key,value> pair
        - Where the "key" is the attribute name
    - Documents in a given collection do not need to have the same structure
      - Flexible schema

7.3.1. MongoDB Components

- A MongoDB datastore may be accessed using the "mongo" shell:
  - It provides a JavaScript-like interface for accessing the MongoDB servers
  - Its "help" command provides a basic description of all the other commands
  - There is also a help() method:
    - "db.help()" describes all methods to be used onto a database "db"
    - "db.col.help()" describes all methods that may be used onto a database collection "col"
- In the simplest configuration, "mongo" interacts with a single "mongod" server
  - The "mongod" server should be already started

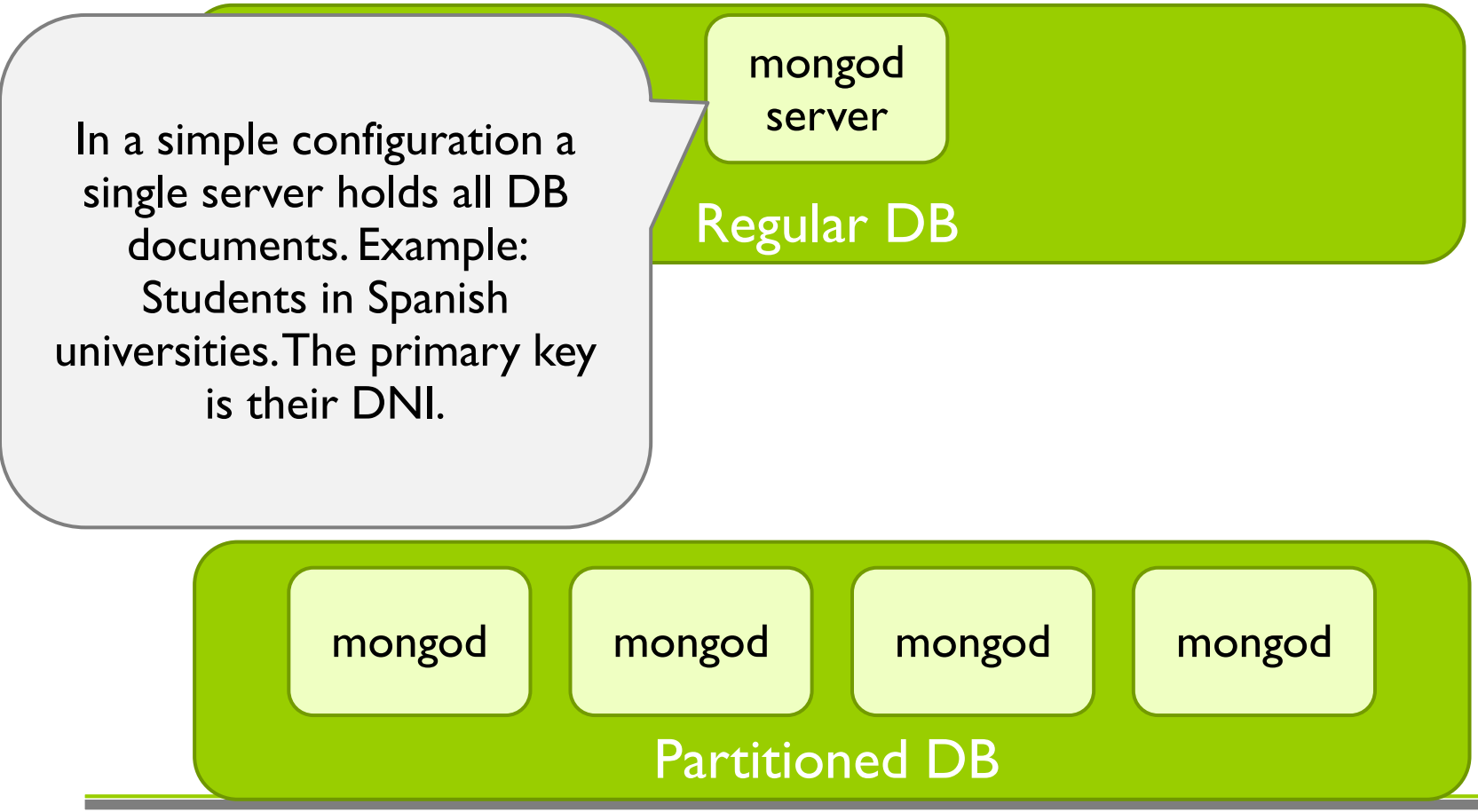
7.3.1. MongoDB Components

- Simple configuration:
  - A single computer
  - with a "mongod" server that manages the database
  - and a "mongo" shell in order to access the DB
    - Or a program written in any of the programming languages with a MongoDB driver



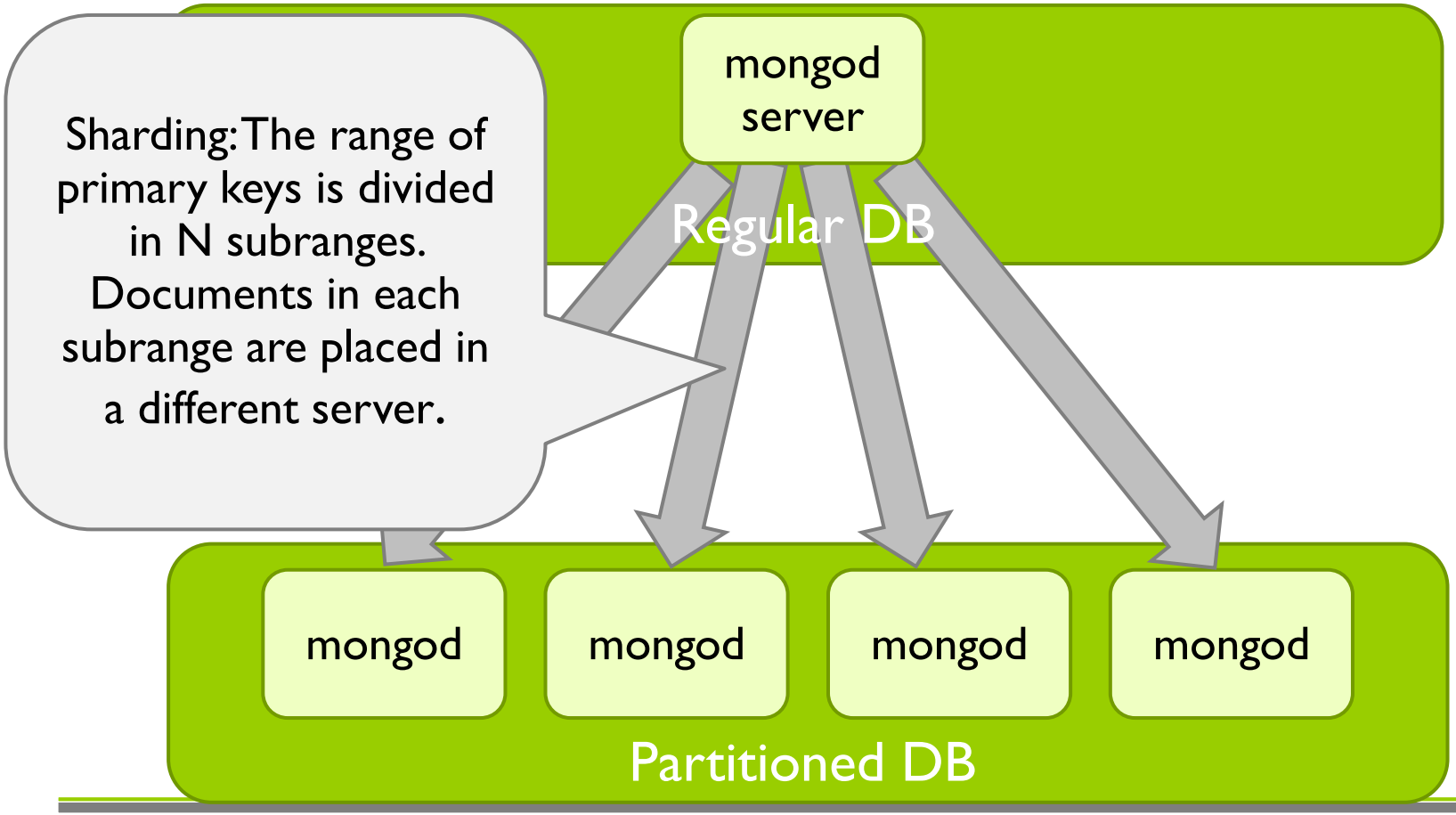
7.3.1. MongoDB Components

- Scalable systems require more complex configurations.
- They need horizontal partitioning (or "sharding")



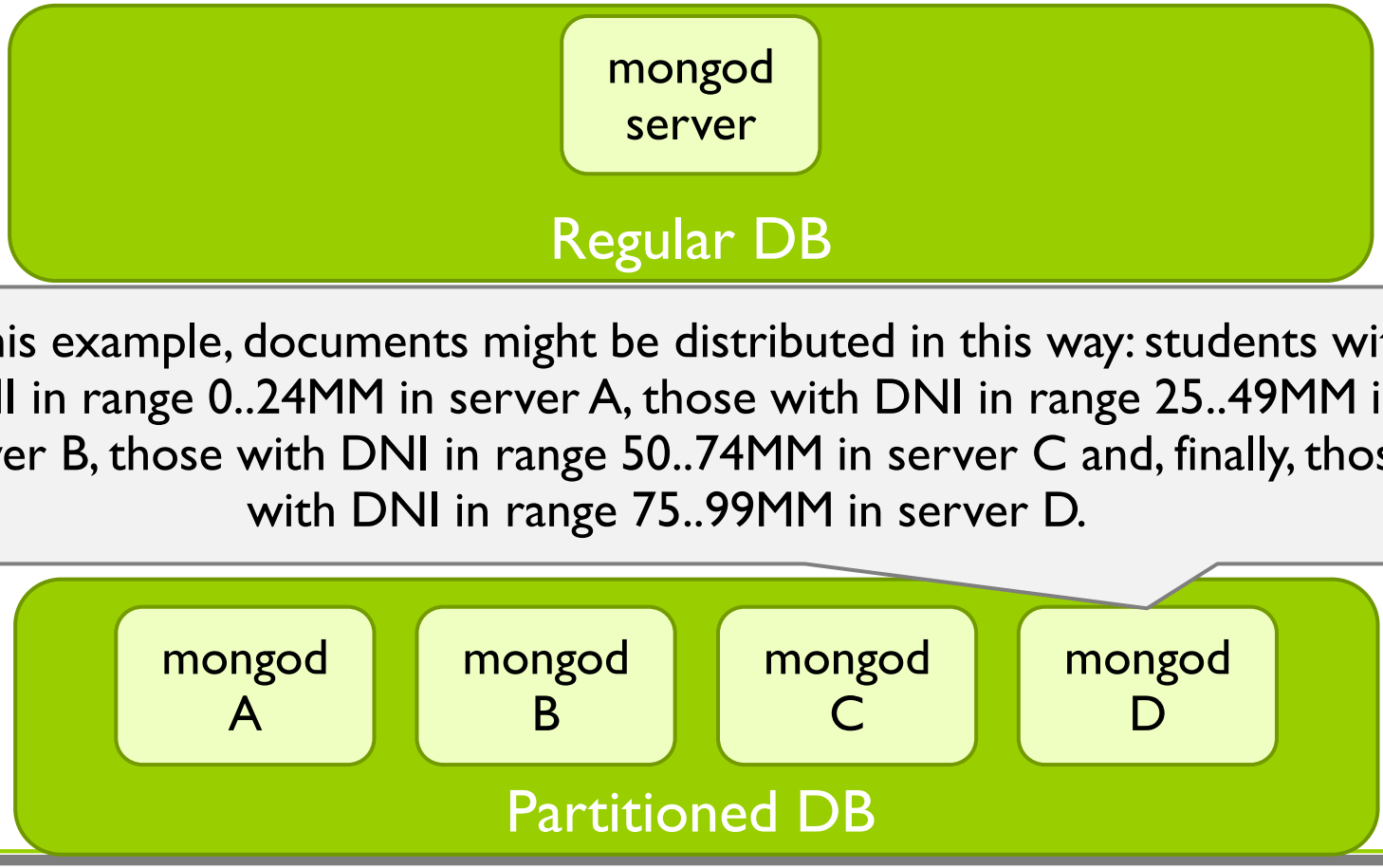
7.3.1. MongoDB Components

- Scalable systems require more complex configurations
- They need horizontal partitioning (or "sharding")



7.3.1. MongoDB Components

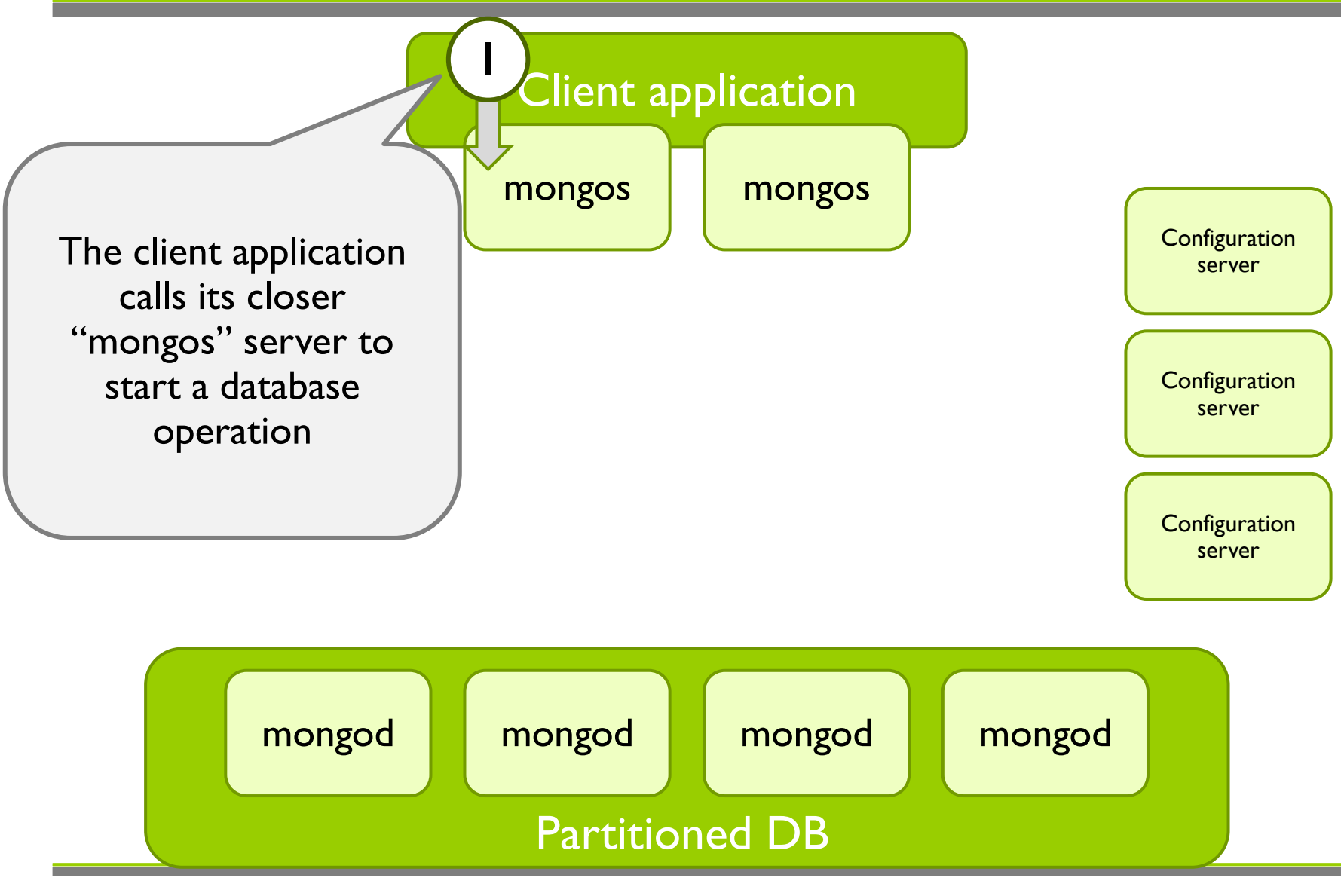
- Scalable systems require more complex configurations
- They need horizontal partitioning (or "sharding")



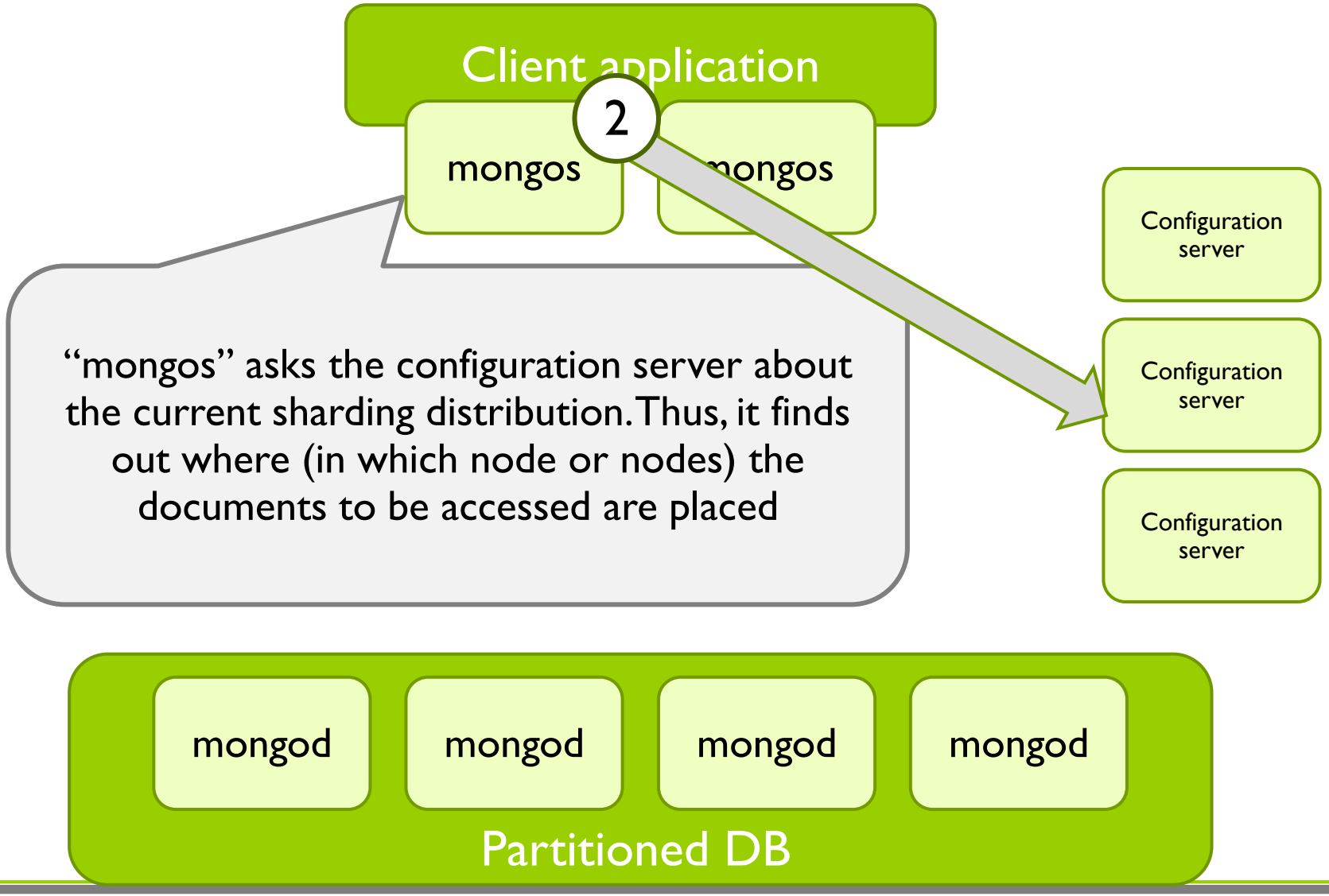
7.3.1. MongoDB Components

- When a large database is being used, three types of servers are needed (usually, with a node per server):
  - "mongod" processes
    - Each one holds a subset (or "shard") of the database documents
    - In order to improve the scalability, horizontal partitioning is used
    - Each "shard" may be replicated
  - "mongos" processes
    - If sharding is used, "mongos" servers behave as the DB interface with the client application
      - They route the requests to the appropriate "mongod" server
    - They query the configuration servers about which ranges are stored in which "mongod" server
      - Later, they forward the requests to the correct "mongod" instance
  - Configuration servers
    - They store the database metadata
    - They know which documents belong to each "shard" and which "mongod" servers manage those "shards"
    - They form a "replica set"; i.e., they use the common replication protocol of MongoDB.

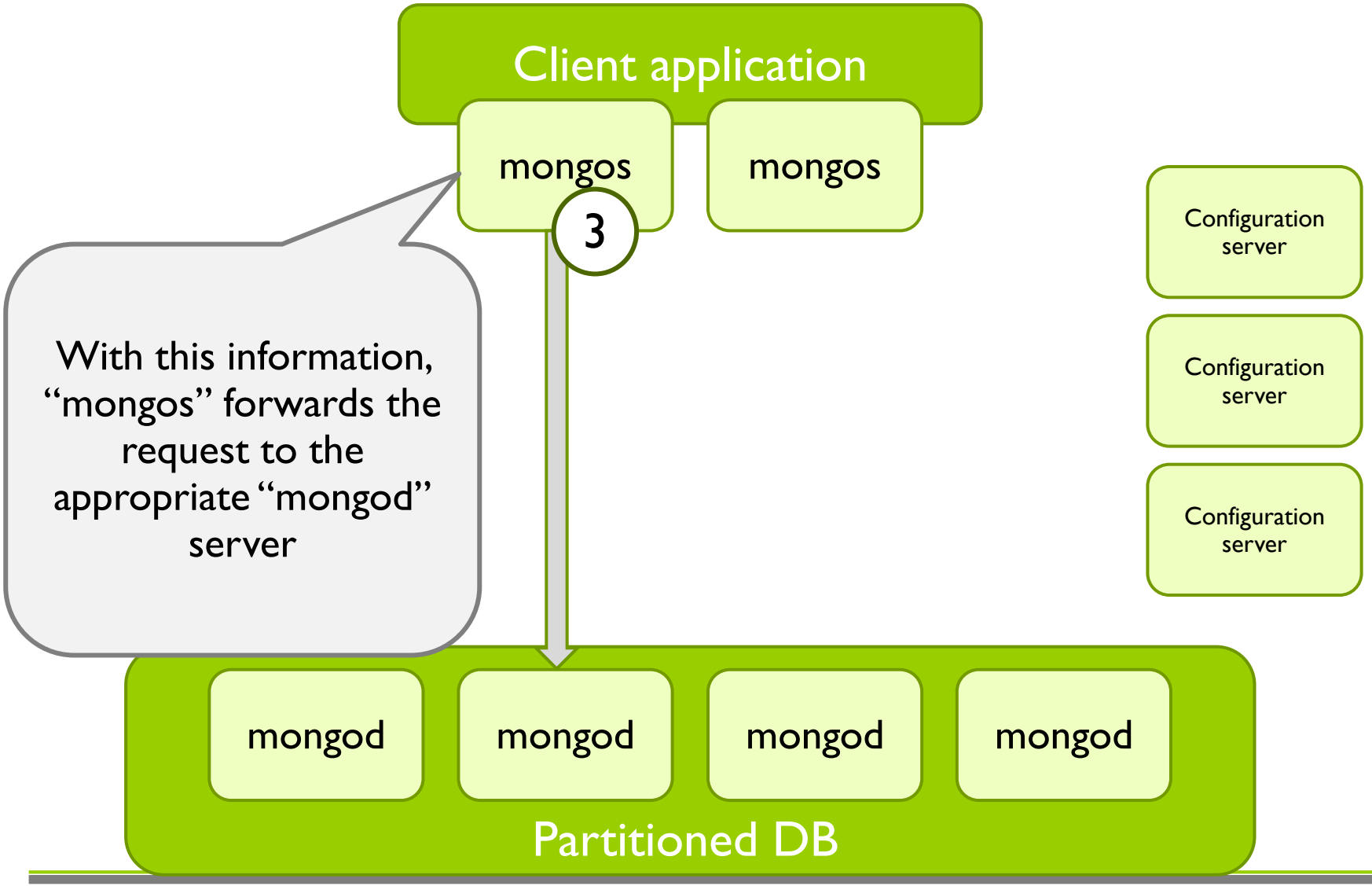
7.3.1. MongoDB Components



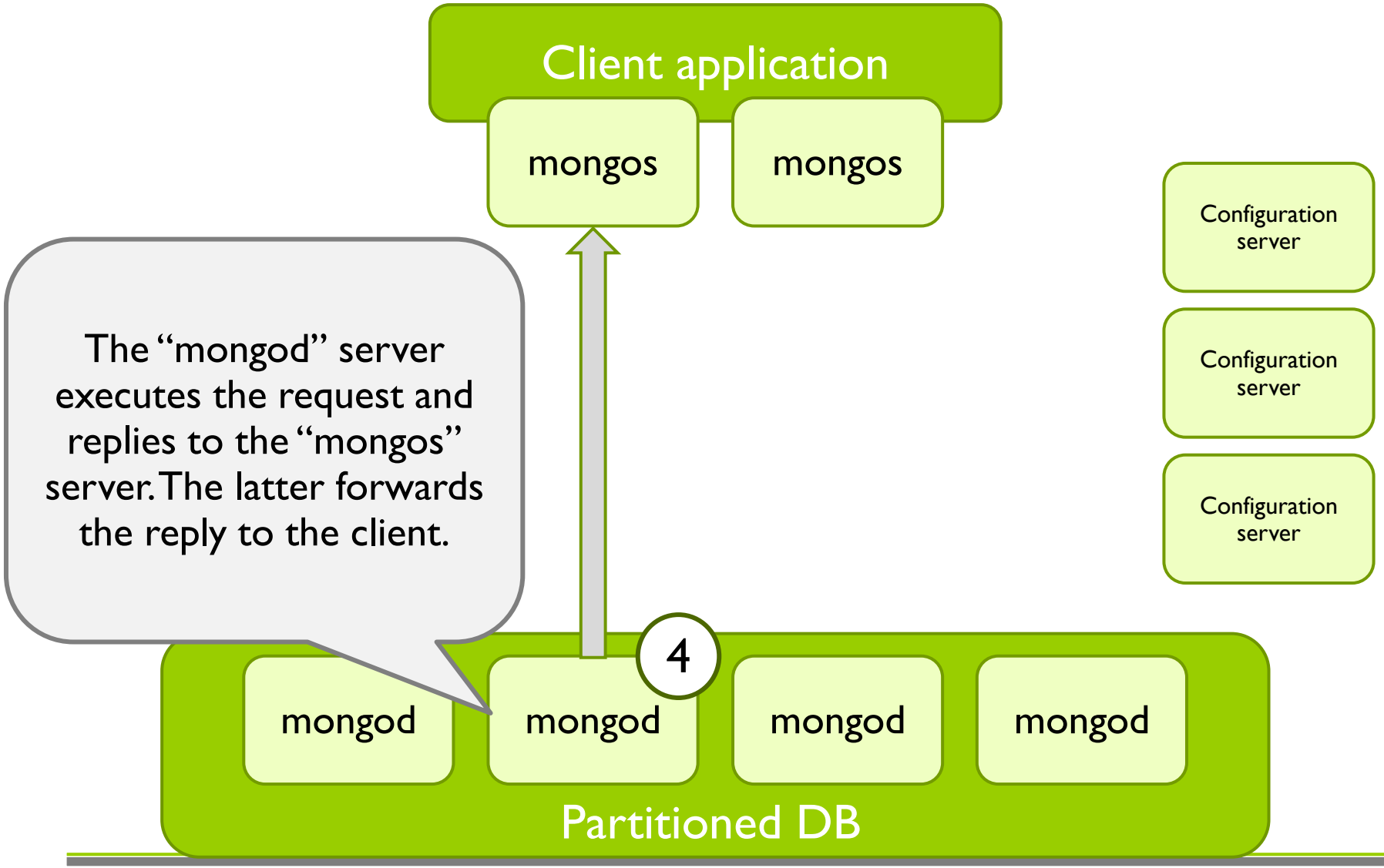
7.3.1. MongoDB Components



7.3.1. MongoDB Components



7.3.1. MongoDB Components



7.3.2. "mongod" Server

- It is the DBMS (DataBase Management System) server
- When no partitioning nor replication exists, a single "mongod" server is enough
- DB partitioning is needed when:
  - A single "mongod" cannot manage the workload, or...
  - The stored data fills the disk drives of the node where "mongod" is executed

7.3.2. "mongod" Server

- When the database is partitioned:
  - The subset (i.e., "shard") of each server is fragmented in "chunks"
  - Each chunk cannot be greater than 64 MB
    - If so happens, the chunk is divided in two halves
  - A load balancing process monitors how many chunks are in each "mongod" server
    - If those numbers are unequal, at least one chunk is migrated from the most "loaded" server to the least loaded one
      - Until the load of every server is balanced
    - In order to migrate a chunk:
      - First, a copy action to the target server is started
      - While the copy is being made, all accesses are still forwarded to the source server
      - When the copy is terminated:
        1. The configuration servers are informed about the end of this migration
        2. The chunk is erased from the source server

7.3.2. "mongod" Server

- In order to avoid inconsistencies, a "*journaling*" mechanism is used:
  - Each update is written first to the "*journal*".
    - The operation type, the elements to be updated and the operation result are written down
    - Later, the update is applied to the database
  - If a failure happens within the update operation
    - The client does not receive any reply
    - When the execution is resumed, the journaling files are scanned, checking whether any operation was not applied to the database
      - Every unfinished update is now applied to the database
  - Advantages:
    - Data corruption is avoided in case of failure
      - There will not be any uncompleted update
      - Database recovery is very fast

7.3.3. "mongos" Server

- It is the unique type of server that may be directly used by client applications when the database is partitioned
- "mongos" does not maintain any DB data element
  - Data are managed and stored by the "mongod" servers
- It behaves as a request router / forwarder
- Metadata on the sharding distribution should be obtained from the configuration servers
  - Once obtained, it is held in a local cache
  - Such cache is refreshed when its contents are unable to find the requested documents
  - Load balancing implies a redistribution of chunks, invalidating some cache contents

7.3.4. Configuration Servers

- They are "mongod" processes.
  - But they do not store DB documents
  - They hold the metadata: shard distribution
- Other characteristics:
  - They are replicated
    - Located in different computers
  - This allows service continuity in case of failures
    - To this end, a "majority" of configuration servers must remain alive
  - Metadata updates are applied using a two-phase commit (2PC) protocol
    - This ensures strong consistency
    - But 2PC is a heavy protocol that needs many messages

7.3.5. Replication

- "mongod" servers may be replicated to:
  - Improve availability
    - Failures are overcome in a transparent way
  - Increase throughput
    - Queries are distributed among replicas
      - Scalable service
- A passive replication model is used
  - The primary replica is the single one that executes inserts, updates and deletes
    - Their result is propagated to the other replicas, that should apply them
    - Such propagation is asynchronous
  - Backup replicas may also execute queries
    - Without any collaboration from the primary replica
- Transparency: Any "mongod" server may be replaced by a "replica set"
  - Interesting for "sharding": Each "shard" may be managed by a different "replica set"

7.3.5. Replication

- Roles
  - Three roles are distinguished:
    - Primary: Replica that manages writing operations
      - Regularly, it also manages queries
    - Backup: Other replicas that hold copies of the database (or shards)
      - They may manage queries
    - Arbiter: Logical replica that does not store any data
      - It votes in the elections for primary in case of failure
  - A replica set cannot have more than 50 nodes
    - The number of "voters" in case of failure is limited to 7
      - The other replicas are "secondary without vote"

7.3.5. Replication

- Failure management
  - Network partitioning cannot be managed
  - In case of failure, MongoDB only goes on when a majority of correct replicas remains active
    - Correct replica -> One that remains active and is able to communicate with the other active replicas in a "majority" subgroup
    - Recommendation:
      - To have an odd number of replicas
        - E.g., 3 replicas overcome 1 failure, 5 replicas overcome 2 failures, 7 replicas overcome 3 failures, …
      - If that number should be even, an "arbiter" replica will be added
        - In case of network partition, it allows to choose a majority group
          - E.g., with 6 replicas and 1 arbiter… a network partition may generate two isolated groups:
            - A: 3 replicas
            - B: 3 replicas + arbiter
            - Only B is allowed to continue, since it maintains a majority of nodes (4 out of 7)
  - Failure detection and primary promotion are managed automatically by MongoDB
    - This is transparent for programmers and users

7.3.5. Replication

- Write synchrony: "*write concern*"
  - Insert, update and delete operations admit a "w" ("write concern") option:
    - It specifies how many replicas should confirm the completion of such writing action before returning control to the invoker
    - By default, its value is 1
      - Value -1 or 0, complete asynchrony
        - Value -1 implies no care at all
        - Value 0 reports, at least, if there have been errors in server communication
        - None guarantees that the primary has completed the write action: unguaranteed persistency
        - Too much dangerous if failures arise
      - Other values:
        - 'majority':
          - Implies value 1 when there is no replication
          - It is a majority of voting replicas in case of replication
          - This is the recommended value!!
        - Any concrete value
          - May introduce problems in case of multiple failures
          - We need to know the initial number of replicas in order to set a reasonable value. Transparency loss!!!
          - Problematic: If there are no such live replicas, the connection is hung out!

Index

8. Learning Results

- At the end of this unit, the student would be able to:
  - Know the "CAP Theorem" and its implications in the design of scalable distributed systems
  - Identify and use several mechanisms for developing scalable distributed services
    - Relaxed consistency
    - Multi-master replication
    - Data partitioning
  - Identify the key aspects of an elastic distributed service.
  - Know and apply some design principles to avoid contention in scalable distributed systems.

Index

9. References

- Wilson, Jim R., *Node.js the Right Way*. Ed. Jacquelyn Carter, Col. The Pragmatic Bookshelf, 2013.
- http://nodejs.org/api/cluster.html (NodeJS Manual & Documentation)
- Ihrig, C., *Scaling Node.js Applications* (http://cjihrig.com/blog/scaling-node-js-applications/), 2013
- Cirkel, K., *Load balancing Node.js* (http://blog.keithcirkel.co.uk/load-balancing-node-js/), 2014
- Robbins, C., *node-http-proxy* (https://github.com/nodejitsu/node-http-proxy)
- Tarreau, W., *HAProxy* (http://www.haproxy.org/) (http://en.wikipedia.org/wiki/HAProxy)
- Sysoev, I., *nginx* (http://nginx.org/en/) (http://en.wikipedia.org/wiki/Nginx)