TSR NOTES 2

UNIT 3: MIDDLEWARE. ZEROMQ

4.4.-ADVANCED SOCKET TYPES

There are two types of **advanced sockets**:

- **Dealer**: similar to a **req** socket, but asynchronous.
- **Router**: similar to a **rep** socket, but asynchronous and with the ability to distinguish its peers.

They are typically found together in the same agent.

4.4.1.-DEALER SOCKETS

It does **not** get **blocked** by failures of peers, but must build a **proper request message**:

- Empty segment (delimiter) before the actual message body.
- Can prepend the delimiter with any number of segments.

Handling a request/reply:

1. Delimiter must be prepended to talk to a rep.
2. When received, the rep socket splits the Envelope (delimiter). The envelope are all the segments up to the first delimiter.
3. App sees only the rest of the message.
4. The envelope is saved by the rep and reinserted to the reply.
5. The dealer application gets all of it.

4.4.2.-ROUTER SOCKETS

It is an asynchronous **bidirectional** socket and allows **sending messages to specific peers**. It assigns an identity to every peer it connects to. The identity is that given to the peer in its program by using “sock.identity = ‘my name’”.

When a **peer has no associated identity socket option**:

1. Router creates a random identity for the connected peer.
2. The created identity lives while the connection is happening.
3. When the connection is cut and re-established the identity changes.

When the router socket passes a message to the app, it prepends an additional segment with the ID of the sending peer.

When the router socket is sending a message. it uses its first segment as a connection identity, thus, a router socket maintains a pair of incoming and outgoing message queues per connection. Such first segment is used to locate the appropriate connection. Once it is found:

- That **first segment is implicitly removed**. The programmer does not need to do anything.
- The **rest of the message is put in the outgoing queue of that connection**. This completes the message sending.

This allows a trivial router-dealer brokering:

- The broker process uses a frontend router socket and a backend dealer.
- Each message received from the router is sent through the dealer.
- Each message received from the dealer is sent through the router.
- In both cases, no message segment needs to be modified.

UNIT 4: SERVICE DEPLOYMENT. DOCKER

1.-CONCEPT OF DEPLOYMENT

A **deployment** is the set of activities that allow a software system to be ready for its use. Activities related with thee installation, activation, upgrade and removal of components or the whole system.

A **distributed application** is a collection of heterogeneous components spread over a computer network. Many components are built by **different developers**. Those components **may change in a fast and independent way**.

Application components should cooperate, so there are **dependences among them**. Those nodes may be heterogeneous, but each component has its **own execution requirements**. There may be **additional security requirements**.

Manual deployment:

- Copy the source code of each component in those computers where each instance should be run. We should guarantee that in those nodes the software base is correctly installed in the proper version.
- Launch every instance of each component in the appropriate order. In the command line needed for starting each component, the appropriate arguments should be stated.

2.-SERVICE DEPLOYMENT

We develop distributed applications in order to provide services (functionality) to remote clients:

Application + Deployment = Service

Every service sets an SLA (**Service Level Agreement**):

- **Functional definition**.
- **Throughput**.
- **Availability**.

Service deployment is an **installation**, **activation**, **upgrade** and **adaptation** of the service:

- **Installation and activation**: software execution.
    - Software dependence resolution.
    - Software configuration.
    - Determine the amount of instances of each component and their distribution in different nodes.
    - Agent dependence resolution.
    - Set the appropriate component start order.
- **Deactivation**: stop the software system in an ordered way.
- **Upgrade**: replace components.

- **Adaptation**.
    - Failure/recovery of an agent.
    - Agent configuration changes.
    - Scaling.

3.-DEPLOYMENT AUTOMATION

A large-scale deployment cannot be manually managed, thus, an **automation tool** is needed. The are **two types of configuration**:

- **Configuration of every component**:
    - File with a list of configuration parameters and dependency descriptions.
    - The tool creates a specific configuration por every component instance.
- **Global configuration plan**:
    - Inter-component connection plan.
    - The location of each instance is decided.
    - Dependency resolution or binding.

There are two options to **dependency resolution**:

- The program code **defines how dependences must be resolved**.
- **Dependency injection**: the program code exposes local names for relevant interfaces. The container fills those variables with object instances. This generates a graph with all service component instances. The edges of the graph are links dependency-endpoint.

4.-DEPLOYMENT IN THE CLOUD

The **deployment in the cloud in IaaS** is based on **hardware virtualisation**. Virtual machines have different sizes, so it exists **flexibility in the resource allocation**. However, there are some **deployment limitations**:

- **Allocation decisions cannot be automated** (low level).
- **No choice on networking characteristics**.
- **Inaccurate failure models**. Not independent failure modes. Limited help in recovering failed instances.

The **deployment in the cloud in PaaS** has the **SLA** as the **central element**, thus, every component has the SLA parameters. This kind of deployment tries to **achieve deployment automation**, but nowadays, it is still with a **limited automation**.

5.-CONTAINERS

Provisioning is booking the infrastructure needed by a distributed application. Resources are needed for component intercommunication.

Specific resources are needed by each instance, so two alternatives appear:

- Instances on **VM** (OS + libraries).
- Instances on **containers** (libraries).

The usage of **containers** presents the following characteristics:

- **Lower flexibility**. Software in each instance must be compatible with the host OS. The container isolation is not perfect.
- **Uses fewer resources**.
- **Easier deployment**.
- **May be applied in many scenarios**.

Docker characteristics:

- Its Dockerfile configuration file automates the deployment of every instance.
- It supports a version control system (Git).
- Besides the native file system, it defines a read-only file system that may be shared among containers.
- Allows cooperation among developers through a public repository.

6.-DOCKER

The **Docker ecosystem** consists of three components:

1. **Images (builder component)**: read-only templates taken as a basis for container instantiation.
2. **Containers (executor component)**: created from images. They maintain all items needed for executing an instance.
3. **Repository (distributor component)**: global store where images can be pushed and pulled.

A **new image** is a base image plus instructions. Docker uses console commands and its general structure is: ***docker*** *action options arguments*. Examples:

- **Information on locale images**: *docker images*.
- **Information on a given image**: *docker history imageName*.

To **create and start a container from an image** we use: *docker run options image initialProgram*. Now, we may modify the container state using command in an interactive session at the console. Finally, we may create a new image **saving the current state of the container**: *docker commit containerName imageName*.

Command groups:

Group	Description
config	Configuration management
container	Operations with containers
context	Contexts for distributed deployment (k8s, …)
image	Image management
network	Network management
service	Service (i.e. multiple containers that run the same image) management in distributed deployments (e.g. with docker-compose or docker swarm)
system	Global management
volume	Secondary storage management

Main commands:

1. **Life cycle management** (in the container group): *docker **commit | run | start | stop***.
2. **Informative** (multiple groups): *docker **logs | ps | info | images |history***.
3. **Repository access** (image group): *docker **pull | push***.
4. **Other** (container group): *docker **cp | export***.

7.-IMAGE CREATION

There are two alternatives for **image creation**:

- **Interactively**:
  - Choose a base image and start a container: *docker run -i -t baseImage initialProgram*.
  - Modify interactively the container state.
  - Save the current state as a new image: *docker commit containerID newImageName*.
- **Create a new image from the instructions saved in a Dockerfile text file**: *docker build -t newImageName pathToDockerfile*.

In a **Dockerfile**, each line starts with a command (in uppercase, by convention):

- The first instruction (in the first line) must be **FROM imageBase**.
- **RUN command** runs that command in the container shell.
- **ADD source destination** copies files from a source (URL, directory or file) to a path in the container. If source is a directory, then it copies all its contents. If it is a compressed file, then it is extracted in the destination.
- **COPY source destination** is equal to ADD, but it does not extract compressed files.
- **EXPOSE port** states the port number where the container listens to incoming requests.

- **WORKDIR path** specifies the container working directory for subsequent **RUN**, **CMD**, or **ENTRYPOINT** commands.
- **ENV variable value** assigns value to an environment variable that may be used by the programs to be run in the container.
- **CMD command arg1 arg2 ...** provides default values for the command and arguments to be run in the container.
- **ENTRYPOINT command arg1 arg2 ...** runs that command when the container is started (and the container is terminated when that command ends).

There must be a single **CMD** or **ENTRYPOINT** in the Dockerfile, otherwise, only the last one is used.

8.-MULTIPLE COMPONENTS IN DIFFERENT NODES

Docker-compose only manages **components in a single host**. If we need multiple hosts, then, the best option is **Kubernetes**. It is a container orchestrator, but it does not depend on Docker. Its **main elements** are:

- **Cluster** and node (real or virtual).
- **Pod**: smallest deployment unit. It includes containers that share a namespace and storage volumes.
- **Replication controllers**: manage the life cycle of a group of pods, ensuring that a specified number of instances is running, scaling, replicating and recovering pods.
- **Deployment controllers**: upgrade the distributed application.
- **Service**: defines a set of pods and their public access.
- **Secrets** (credentials management).
- **Volumes** (persistence).

UNIT 5: FAILURE MANAGEMENT

1.-INTRODUCTION

There is a **failure** when any system component is unable to behave according to its specification. **Three different concepts** should be identified:

- **Fault**: anomalous physical condition.
- **Error**: manifestation of a fault in a system.
- **Failure**: a failure occurs when an element is unable to perform its designed function due to errors in the element or its environment, which are caused by various faults.

When a fault is not detected nor handled, it can become an error. When errors occur without redundancy, they can become failures.

A **distributed system** should **provide failure transparency**, so when errors in components happen, should not be perceived by users. The **solution** to this problem is **replication**. A faulty replica should be diagnosed and separated and once repaired, it will rejoin the system. Other replicas hide all these steps.

This transparency is also known as "**fault tolerance**". A system is fault-tolerant when it behaves correctly in case of faults. In order to be fault-tolerant, a **service needs**:

- To be carefully designed.
- That all external services it depends on be also fault-tolerant.

Failures may have multiple causes. They depend on faults and on which elements are affected by those faults. Multiple kinds of failures can be distinguished.

A given **failure model** should be assumed when a distributed algorithm is written. Models cannot include all failure scenarios, but they are a higher level of abstraction. The concrete details are not considered. The middleware should translate the physical scenario into a logical one that matches what is considered in the assumed failure model.

There also exist **network failures**, so the connectivity should be considering. **Network partitioning** is a problem. There are **two options** to solve this problem:

- **Partitionable system**: each isolated subgroup is able to go on. Some kind of reconciliation protocol is executed when connectivity is recovered.
- **Primary partition model**: only the group with a majority of system nodes is able to go on. Sometimes, there is no such group.

A **partition** is when a group of nodes remains isolated from the other parts of the system.

2.-REPLICATION

Replication is a basic mechanism in order to ensure component **availability**. Each component replica is placed in a different computer. Those computers do not depend on the same failure sources. When a **replica fails the others will not fail**, thus, on-going operations in the crashed replica may be resumed in any other replica.

Replication makes **easy** the **recovery** of failed replicas since correct replicas are the source of a state transfer once a crashed replica (is repaired and) rejoins the system.

Replication also **improves service performance**:

- **Read-only operations may be served by any single replica**. Then, there is a linear throughput that depends on the number of replicas. Excellent for scalability.
- **Update operations need to be applied in all replicas**. This introduces non-negligible delays. Short operations may be executed in all replicas and requests should be propagated to all replicas. Long operations that update a few data elements may be executed in a single replica, propagating later the updated elements to all other replicas. Operations may be executed (or update-propagated) in different order in different replicas, thus, the result state may diverge among replicas. Such degree of allowed divergence determines a consistency model.

There are **two classical replication models**:

- **Passive** (or "primary/backup replication"): clients send their requests to a single primary replica. Such replica executes all the operations. Once an operation is concluded, its updates are propagated to the secondary (or back-up) replicas and it answers the client.
- **Active** (or "state-machine replication"): each client request is propagated to all server replicas. Every server replica executes such operation. When a replica concludes the operation, it replies to the client.

The **advantages of passive replication** are:

- **Minimum overhead**: each request is served by a single replica. Read-only requests might be served by any secondary replica, then having an efficient load balancing.
- **Updates can be easily ordered**: the primary replica tags the updates with a sequence number before broadcasting them to secondary replicas.
- **Local concurrency control**: no distributed algorithm is needed.
- **Admits non-deterministic operations**. They are only executed by the primary. Inconsistencies among replicas never arise

The main **inconveniences of passive replication** are:

- **Difficult reconfiguration** when primary fails since a secondary must be chosen and promoted to primary. Ongoing requests might be lost.
- **Arbitrary failures cannot be managed**.

The **advantages of active replication** are:

- **Trivial recovery in case of failure** as nothing needs to be done. Assuming that at least a replica survives.
- **Allows arbitrary failures**, assuming a maximum of "f" simultaneous failures, at least 2f+1 replicas are needed. Client waits for at least f+1 identical answers.

The main **inconveniences of active replication** are:

- **When a strong consistency is needed, requests should be delivered in total order**. This needs consensus, but they are expensive (multi-round) broadcast protocols.
- **Non-deterministic operations are difficult to manage**. Each replica might obtain a different result and this leads to inconsistencies.
- When several actively replicated servers interact, **requests need to be filtered**.

Comparison summary:

Aspect	Passive model	Active model
Request processing replicas	**I**	All
Avoids distributed request ordering among replicas	**Yes**	No
Avoids update propagation	No	**Yes**
Admits indeterminism	**Yes**	No
Tolerates arbitrary failures	No	**Yes** (However, to this end, even read only operations should run in all replicas)
Consistency	At least, sequential	At least, sequential
Recovery in case of failure	It needs election and reconfiguration stages in case of primary failure	**Immediate**

The replication model to be chosen depends heavily on:

The average request processing time. The **passive model** is appropriate for **long processing times**. Since that processing is only done at a single replica. The **active model** appropriate for **short processing times**.

The updates size. The **passive model** appropriate for **small updates** and the **active model** appropriate for **big updates**, since it avoids update transfer.

3.-CONSISTENCY

When no synchronization tools are assumed, the **main consistency models** are:

- Strict (slow).
- Sequential (slow).
- Processor (slow).
- Cache (slow).
- Causal (fast).
- FIFO/PRAM (fast).

There also exists **eventual consistency**. In scalable systems, replicas converge when there are long time intervals without any update operation. When there are pending update operations, each replica serves them without worrying about what other replicas are doing. Later on, processes decide how to interleave those updates and which final value should be adopted. It is trivial if operations are commutative.

3.1.-STRICT CONSISTENCY

Assumes a **global clock that timestamps each event**:

- Two writes cannot happen simultaneously.
- Writes are propagated immediately.
- Each read event on variable x returns the value of the closest earlier (by the timestamp order) write event on variable x.

It is difficult to implement since it implies a strong coordination among all replicas. It is not scalable.

3.2.-SEQUENTIAL CONSISTENCY

All processes reach a consensus on the order of all writes. This order matches all process-writing orders, but each process advances at its own pace.

3.3.-CAUSAL CONSISTENCY

Complies with the “happens before” relation defined by Lamport, relation used in logical clocks. Causal consistency assumes the mapping we introduced earlier: W(x)a -> R(x)a.

3.4.-FIFO/PRAM CONSISTENCY

Requires that all writes made by each process are read in writing order by all other processes, but there is no constraint for interleaving what has been written by different processes.

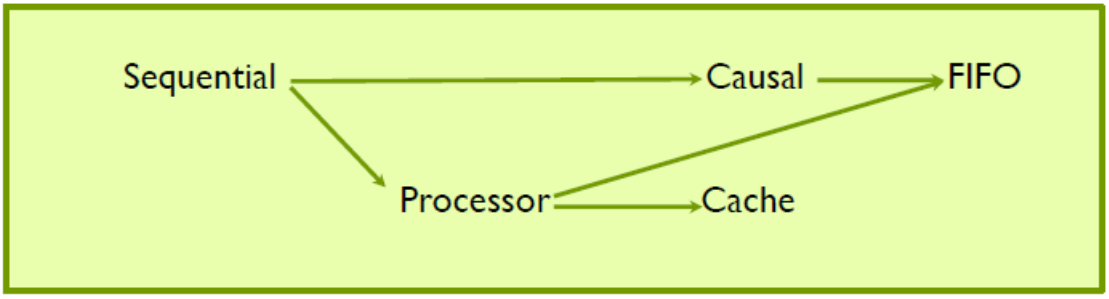
3.5.-CACHE CONSISTENCY

This model requires that writes onto each variable are seen in the same order by all processes. Additionally, when a given process writes multiple times onto a variable, these writes are seen in such write order by all processes, but there is no constraint in order to interleave reads made on different variables.

3.6.-PROCESSOR CONSISTENCY

We obtain “processor” consistency when both cache and FIFO consistencies are respected.

3.7.-MODEL



UNIT 6: SCALABILITY

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.
- **Data can be distributed among multiple servers** by using data partitioning techniques, like NoSQL datastores.

2.-CAP THEOREM

In a distributed system these three properties cannot be ensured simultaneously (CAP Theorem):

- **Strong consistency (C)**.
- **Service availability (A)**: 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-partition tolerance (P)**. A network-partition is a situation which implies that at least a system subgroup becomes unable to communicate with all the other subgroups, arising a loss of connectivity.

One of these properties should be sacrificed.

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.

When **strong consistency and full service availability need to be maintained, partition tolerance is dropped**. Then, we will try to ensure connectivity, for instance, replicating the network. But continuous connectivity is hard to ensure, and this is only affordable in local deployments involving a single lab, and even in that case it is extremely difficult to ensure.

When we **renounce to availability, 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, but 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, so all their nodes can maintain an internal strong consistency.

When we **renounce to consistency, 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, then, the strong consistency is lost. A carefully designed system, with commutative update operations, ensures eventual consistency using trivial reconciliation procedures.

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, so 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**.

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, but a third solution exists: **multi-master replication**. It is 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 and updates are forwarded in a lazy way to the remaining replicas.

The **main advantages** are:

- **Minimum overhead**: each request is served by a single replica for both read-only and update requests.
- **Highly scalable**, since no concurrency control mechanisms are assumed.
- It **admits non-deterministic operations**. They are only executed by a master. Per-operation inconsistencies among replicas never arise.

The **inconveniences** are:

- **Problems in case of failure** of a master since ongoing requests might be lost.
- **Inherits the weak failure model management of the passive model**, so 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.

4.-NOSQL STORES

Approaches to **improve database scalability**:

- **Simplify the schema** by replacing a set of relational tables with simple key-values ones (simple indexes). This simplifies also the query language (immediate processing). The space needed by the database is also reduced, then, databases may be kept in main memory and durability is ensured by replication.
- **Renounce to transactions**. Transactions will no longer group and protect multiple sentences. Atomicity will be limited to each individual sentence, leading to very short blocking intervals (indiscernible).

The **result** of all this are **NoSQL datastores**. There are three 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**. It doesn't have support for distinguishing sub-attributes in the "value" and there is no way for querying about non-primary attributes.

4.2.-DOCUMENT STORES

Schema **composed by objects with a variable number of attributes**. In some cases these attributes may be other objects. The query language is based on setting constraints on the attribute values.

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. It has an easy load balancing, it is highly concurrent and easily scalable to multiple servers.

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.

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.

Elasticity is important in current cloud systems in order to reduce user costs and manage resource provisioning.

It requires:

- **A monitoring system** for the current workload and the current throughput.
- **A reactive system** for automatizing service reconfiguration, adding nodes and other resources when workload is increased and releasing resources when workload decreases. It depends on the current “Service Level Agreement” (SLA). The system reaction should be fast in order to comply with the service level objectives.

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 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.

To **avoid contention**, we use:

- 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**. There is no concurrency and has a trivial management. There is a fast completion of each request. The resulting system is fast and efficient, since it provides less context-switching overhead and lower middleware and/or operating system intervention.
- In **case of a wrong distribution of resources**: **redistribute or replicate resources**. Remote accesses become local accesses.

7.-SCALABILITY EXAMPLES

7.1.-THE “CLUSTER” MODULE IN NODE.JS

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.

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**. It allows creating processes that share the port or ports associated to the service being implemented by that "cluster". It provides a **Worker class** that models the **cluster workers**. They are supervised and created by a "master" process. It provides also some events that may be used for controlling the current state of the worker processes. It uses **IPC** for master-worker communication.

The **events of a Cluster object** are:

- **'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.

There exist **other type of events**:

- **'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().

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.

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**(true when the worker exits voluntarily; false when it has been killed by other processes).
- **Methods**, that are invoked using its process attribute: send() (needed for sending messages to the master process), disconnect(), kill().
- **Events**: 'message', 'online', 'listening', 'disconnect', 'exit', 'error'.

7.2.-SCALABILITY USING MULTIPLE COMPUTERS

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].

7.2.1.-NODE-HTTP-PROXY

It is an **open-source proxy server for Node applications**. Using it, we may **configure http server in multiple machines**. It **balances the workload** among all those server instances.

7.2.2.-HAPROXY

It is **open-source proxy server** (for **TCP and HTTP applications**). Written in C. It provides **load balancing, high availability and scalability onto multiple computers**. It is used in some websites.

7.2.3.-NGINX

It is a **reverse proxy for HTTP and e-mail applications**. As all reverse proxies, it may **configure multiple HTTP servers** in different computers, **balancing their workload**.

7.3.-MONGODB

MongoDB is a scalable "**document datastore**". In a document datastore, the **database** is a **set of collections** (tables). 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 and 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**).

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**.

A **simple configuration** would be 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.

Scalable systems require more complex configurations. They need horizontal partitioning (or "sharding"). In **sharding**, the range of primary keys is divided in N subranges. Documents in each subrange are placed in a different server.

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, and 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".

The **"mongod" server** 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.
- The stored data fills the disk drives of the node where "mongod" is executed.

When the database is partitioned, **the subset 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**:

1. First, a copy action to the target server is started
2. While the copy is being made, all accesses are still forwarded to the source server
3. When the copy is terminated:
    a. The configuration servers are informed about the end of this migration.
    b. The chunk is erased from the source 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.

The main **advantages** are:

- Data corruption is avoided in case of failure.
- There will not be any uncompleted update.
- Database recovery is very fast.

The **“mongos”server** is the unique type of server that may be **directly used by client applications when the database is partitioned**. It 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.

The **configuration servers** are “mongod” processes, but they **do not store DB documents**, they **hold the metadata** (shard distribution). Other would be **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.

“**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.

Any “mongod” server may be replaced by a “replica set”, this provides **transparency**. It is interesting for “sharding” since each “shard” may be managed by a different “replica set”.

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”.

Network partitioning cannot be managed. In case of failure, **MongoDB** only goes on when a **majority of correct replicas remains active**. The **correct replica** is the one that **remains active and is able to communicate with the other active replicas in a “majority” subgroup**. A **recommendation** would be to have an odd number of replicas. If that number should be even, an “arbiter” replica will be added. In case of network partition, it allows to choose a majority group.

Failure detection and primary promotion are managed automatically by MongoDB. This is transparent for programmers and users.

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**, but it can have a different one:

- **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). It is too dangerous if failures arise
- **'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, but this implies a **loss of transparency**. It is problematic if there are no such live replicas, the connection is hung out.