POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea

Fault-tolerance classification in virtualized redundant environment using Docker containers technology

Relatori
prof. Stefano Di Carlo
prof. Alessandro Savino

Candidato
Gennaro Cimmino
matricola: 274493

Anno accademico 2019-2022

A te, nonna ...

Indice

Elenco delle figure

Capitolo 1

Introduzione

With the popularity of cloud computing platforms, server virtualization is a widely used and powerful technique, for consolidating servers in data centers. For many years, standard virtualization enable an efficient sharing of physical resources by multiple, independent entities. Therefore, this technology provide cost reduction, through resource consolidation, and reliability benefits. However, it has several drawbacks: running a virtual machine is often resource-intensive, complex and costly.

An alternative to classical virtualization is the container virtualization. Container virtualization is a virtual run-time technology, that emulates a hosting operating system kernel, rather than the underlying hardware, as in the case of classical virtualization. Nowadays, with the emergence of Docker, containers are gaining in popularity. In fact, a lot of cloud providers, such as Amazon, Google, Azure and Digital Ocean, are leveraging Docker features. Docker is a software platform that allow applications distribution, through containers deployment. Docker containers have many interesting advantages, such as fast start-up, lightweight execution environments, easy deployment and portability. Lastly, Docker provides useful supporting tools, such as the docker-compose. Docker-compose is an orchestration tool that introduce automated configuration, coordination and management of container services. It's used to deploy multi-container applications.

However, the issue of container security and stability is increasingly relevant, especially with the rise of multi-container and large-scale application systems. Many cloud systems depend on the reliability of containers. As a result, a critical failure in these systems, led to invalidation of thousands of services. Therefore, it's necessary to increase the reliability of these systems, by introducing fault tolerance mechanisms. One of the techniques used to

increase the reliability of a system, is redundancy.

Aim of this thesis is to provide methodologies for introducing redundancy, in container systems. A container is redundant if multiple instances of the same, are deployed in a system. Outputs generated by each redundant container, is monitored by a voting system, which guarantees the correct execution of the service. Each methodology introduced, differ according to a containers classification. The classification is based on parameters, needed to introduce additional blocks, suitable for ensuring an entire control system. Classification allows to include,in a docker-compose, all redundancy parameters to containers, that belong to a specific class. In this way, a service can be deployed in fault tolerant mode, using an alternative version of that compose file.

Finally, for the purpose of testing the feasibility of a proposed methodology, a MySQL database container, was deployed in fault-tolerant mode.

Capitolo 2

Docker containers

Docker is a popular open-source engine to perform virtualization mechanism, that automates the deployment of applications into containers. It was written by the team at Docker Inc., formerly called dotCloud Inc., which is an early player in the Platform-as-a-Service market, and released by them under the Apache 2.0 license. Container refers to a lightweight, stand-alone, executable package of a piece of software that contains all the libraries, configuration files, dependencies, and other necessary parts to operate the application.

A container engine uses the Linux Kernel features like namespaces and control groups to create containers on top of an operating system, called *OS-level virtualization*. Docker has well-defined wrapper components that make packaging applications easy. Before Docker, it was not easy to run containers. Meaning it does all the work to decouple your application from the infrastructure by packing all application system requirements into a container.

2.1 Software Virtual Machine

A virtual machine is a computer file or software usually termed as a guest, or an image that is created within a computing environment called the host.

The implementation of a VM emulate specialized software or an entire operating system. Therefore, are broadly divided into two categories depending upon their use:

- *System virtual machines:* also termed full virtualization VMs. They provide functionality needed to execute entire operating systems. These

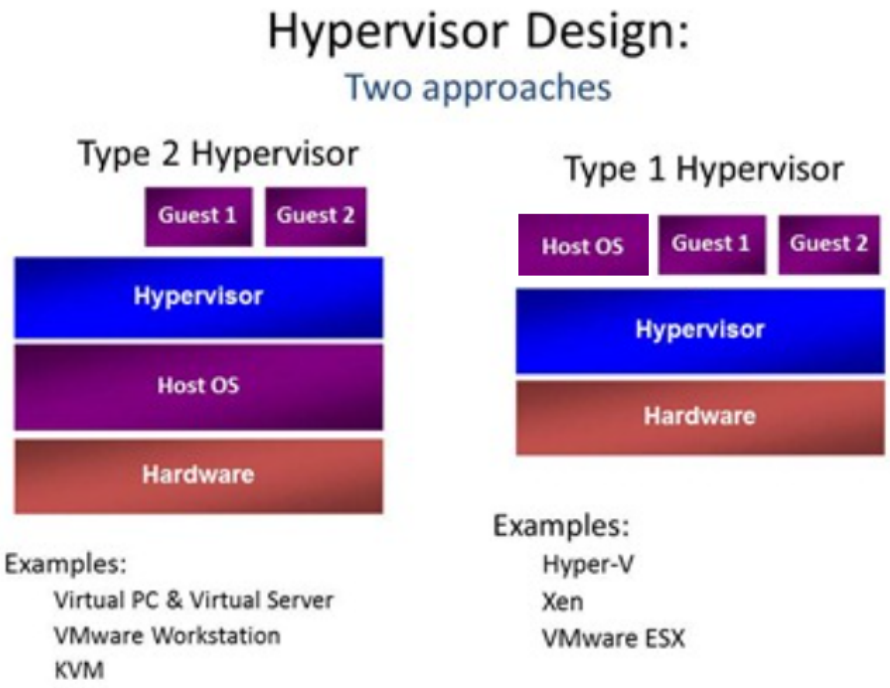

Figura 2.1. Standard Virtualization

types of systems allow sharing resources among virtual machines installed, in order to perform a simultaneous execution of multiple operating systems;

- *Process virtual machines*: allow to a single process to run as an application on a host machine. They are designed to execute computer programs in a platform-independent environment. This feature is obtained by masking the information of the underlying hardware or operating system. An example of a process VM is the Java Virtual Machine, which enables any operating system to run Java applications, as if they were native to that system;

Regarding system virtualization, it uses software to simulate virtual hardware that allows multiple VMs to run on a single machine. This process is managed by software known as a *hypervisor*. The hypervisor is responsible for managing and provisioning resources, for instance memory and storage, from the host to guests. It also schedules operations in VMs so they don't

overrun each other when using resources. VMs only work if there is a hypervisor to virtualize and distribute host resources. There are two types of hypervisors used in virtualization, also showed in fig. 2.1:

- *Type 1 hypervisors or bare metal hypervisors*: The feature of this type of virtualization is that the hypervisors are installed natively, on the underlying physical hardware. A computer, on which the hypervisor runs single or multiple virtual machines, is called a host machine. VMs interact directly with hosts, to allocate hardware resources without any extra software layers in between. Host machines, running type 1 hypervisors, are embedded solutions for virtualization. They're often found in server-based environments, like enterprise datacenters. Some examples of type 1 hypervisors include Citrix Hypervisor and Microsoft Hyper-V. A separate management tool is needed to handle guest activities, like creating new virtual machine instances or managing permissions;

- *Type 2 hypervisors or hosted hypervisors*: they run on the host computer's operating system. Hosted hypervisors forward virtual machine requests to the host operating system, which then provides the appropriate physical resources to each guest. One disadvantage of this type of virtualization is the speed. Compared with type 1 virtualization, it has a higher latency, being that the handling of operations must be handled first by the operating system. Unlike bare-metal hypervisors, guest operating systems are not tied to physical hardware. Users can run virtual machines and use their computer systems as usual. This makes Type 2 hypervisors suitable for personal or small business users, who haven't got dedicated servers for virtualization;

2.2 Container virtualization and VMs

A container is a virtual runtime environment, that runs on top of a single operating system kernel, and emulates an operating system rather than the underlying hardware, as showed in fig 2.2. Instead of using an hypervisor, it leverage features of the host operating system, to isolate processes and control the processes' access to CPUs, memory and desk space. There are several differences between container virtualization and virtual machines. In particular, containers offer:

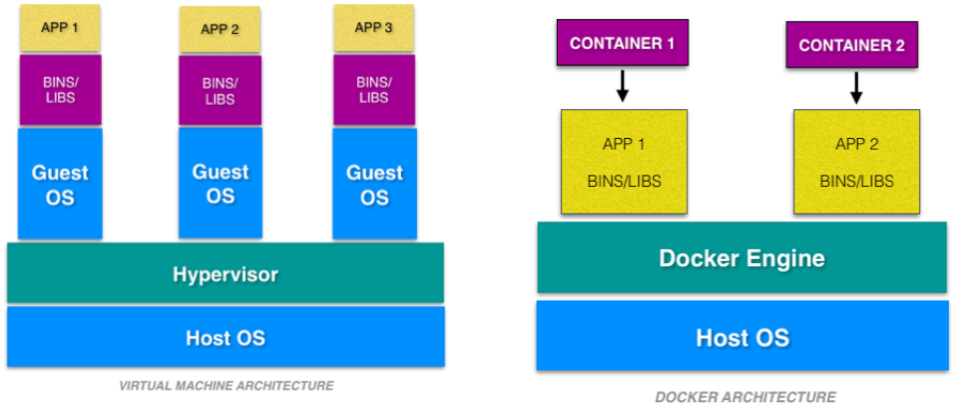

Figura 2.2. Virtualization types

- *Better performance*: Because containers are lightweight and only include high level software, they are very fast to modify and iterate on. Furthermore, a containerized application usually starts in a couple of seconds. Virtual machines could take a couple of minutes;

- *Flexible resource distribution*: Containers use up only as many system resources as they need at a given time. Virtual machines usually require some resources to be permanently allocated before the virtual machine starts. For this reason, virtual machines tie up resources on the host, even if they are not actually using them. Containers allow host resources to be distributed in an optimal way;

- *Direct hardware access*: Applications running inside virtual machines generally cannot access hardware like graphics cards on the host in order to speed processing, but containers can access directly to this resources;

- *Less memory utilization*: VMs are more resource-intensive than containers, as the virtual machines need to load the entire OS to start. Installation of an entire guest operating system, require duplication of a lot of components, already running on the host server. Containers doesn't require all components, but it's necessary to install a minimal amount of libraries;

- *Portability*: With docker containers, users can create an application and store it into a container image. Then, they can run it across any host

environment. VMs don't have a central hub and they requires more memory space to store data;

Container virtualization, despite of all its advantages, cannot replace virtual machine because it could be a better choice in some situations. Virtual machines are considered a suitable choice in a production environment, rather than containers since they run on their own OS without being a threat to the host computer. In fact, it's possible that a break out in one or more container, could affect the shared underlying hardware, since they all share the underlying hardware.

Furthermore, most popular container runtimes have public repositories of pre-built containers. There is a security risk in using one of these public images, as they may be vulnerable to corruptions or manipulations.

Another disadvantage is that containers have a complex usage mechanism and managing tools, whereas tools are easier in VMs to work with.

2.3 Docker containers

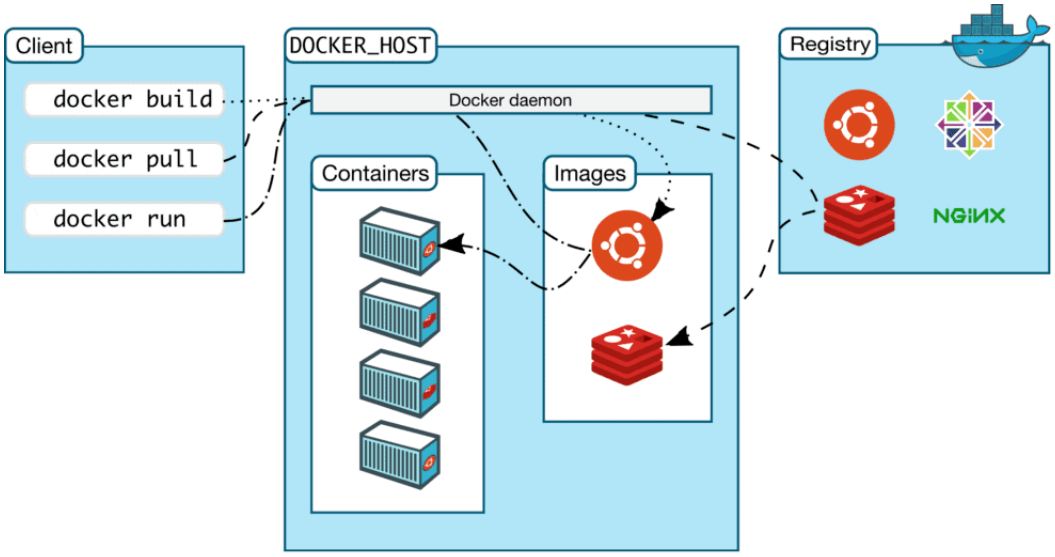

Figura 2.3. Docker system

Docker is a virtualization platform based on containers. It is a container management system that helps to efficiently manage Linux Containers (LXC),

more easily and universally. This lets you create images in virtual environments on a host machine and run commands against them. The actions performed to the containers, running in these environments, locally on a machine, will be the same commands or operations running against them, when they are running in a production environment. Overall, Docker greatly simplifies containers deployment, bringing benefits to container virtualized systems. Some of the main benefits Docker offers are:

- Containers have the added benefit of running anywhere and scalable, thanks to the Docker images feature. A Docker image is made up of a collection of files that bundle together all the essentials components required to configure a fully operational container environment, such as installations, application code, and dependencies;

- Docker improves container isolation by leveraging the *namespaces* feature. Namespaces make sure that a container's filesystem, hostname, users, networking, and processes are separated from the rest of the system. In addition, they also allow isolation between containers, as processes running within a container cannot see, and even less affect, processes running in another containers. Each container also gets its own network stack, meaning that a container doesn't get privileged access to the sockets or interfaces of another container. Container communications are allowed only if the host system is setup accordingly, so they can interact with each other, through their respective network interfaces;

- Another important feature are *cgroups*. They are responsible for managing resources used by a containers. They provide many useful metrics, but also help ensuring that each container gets its fair share of memory, CPU, disk I/O and, more importantly, that a single container cannot bring the system down by exhausting one of those resources;

2.4 Docker engine

Docker is a client-server application. The Docker client talks to the Docker server or daemon, which in turn, does all the work. Docker system has several components, that allow developing, assembling, shipping, and applications deployment. Docker system is also called Docker engine, as showed in fig 2.4, and it has following components:

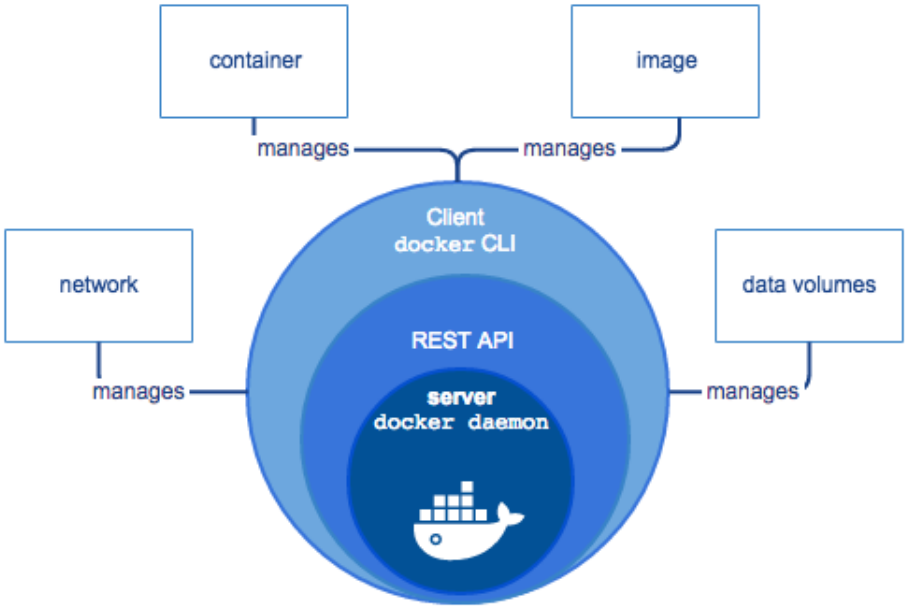

Figura 2.4. Docker engine

- *Docker Daemon*: A persistent background process that manages Docker images, containers, networks, and storage volumes. The Docker daemon constantly listens for Docker API requests and processes them;
- *Docker Engine REST API*: Communication between Docker daemon and applications, pass through a RESTful API. It can be accessed by an HTTP client;
- *Docker CLI*: A command line interface execute commands, like Docker build, Docker run, and so on, and handle interactions with the Docker daemon;

The main objects for application development are two: Docker images and Docker compose files.

A Docker image is a read-only template that contains a set of instructions for creating a container, that can run on the Docker platform. Each of the files that make up a Docker image is known as a layer, as showed in fig 2.5. These levels are a series of intermediate images, built on top of each other in successive stages, where each level depends on the one immediately below.

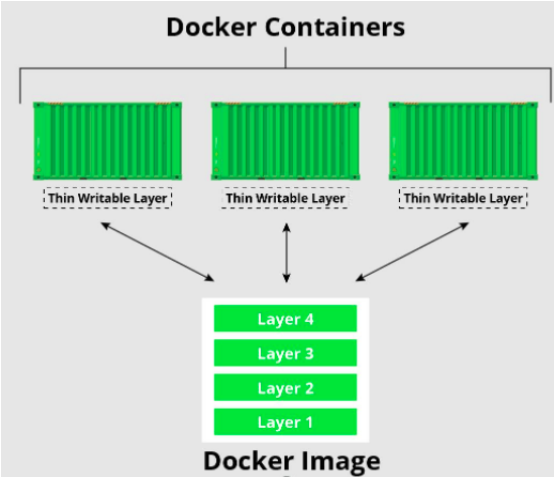

Figura 2.5. Docker Images

Thanks to this architecture, changing a layer at the top of a stack, require a minimum amount of computational work, in order to rebuild the entire image. This is because, Docker doesn't need to rebuild lower layers, but only top layers modified.

Docker Compose is used for running multiple containers simultaneously, using a single command. All containers are defined within a compose, written in a scripting language called YAML. It's a XML-based language, that stands for "Yet Another Markup Language". These tools that automate simultaneous configurations, coordination, and management of systems , are called *"Orchestrators"*.

Capitolo 3

Redundancy

3.1 Fault-Tolerance

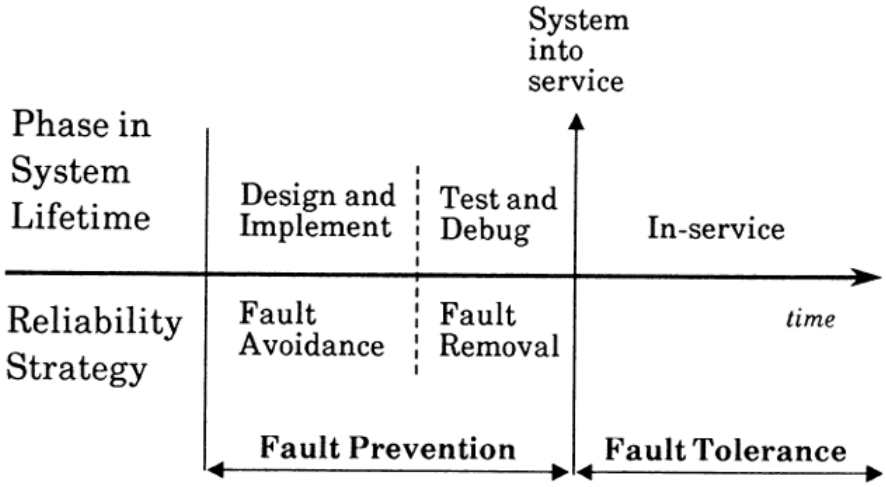

Figura 3.1. Reliability scheme of systems

Continuous reduction of feature sizes implies that, both transient and permanent hardware faults are more probable in today's integrated circuits. There are two types of failures: transient errors, or soft errors, and permanent errors, or hard errors. Atmospheric disturbances and natural radiations may lead to soft errors, whereas manufacturing defects or extreme operating conditions can cause hard errors. In order to provide reliable computing systems there are two complementary approaches which can be adopted, as shown in fig 3.1: fault prevention and fault tolerance. The former approach is more

straightforward, as it tries to ensure that the system is error free. In this way, the reliability of the system is increased, ensuring that new failures cannot occur and potential faults are avoided.

The second approach accepts that an implemented system will not be perfect, and that measures are therefore required to enable the operational system to cope with the faults that remain or develop.

3.1.1 Abstraction levels of fault model

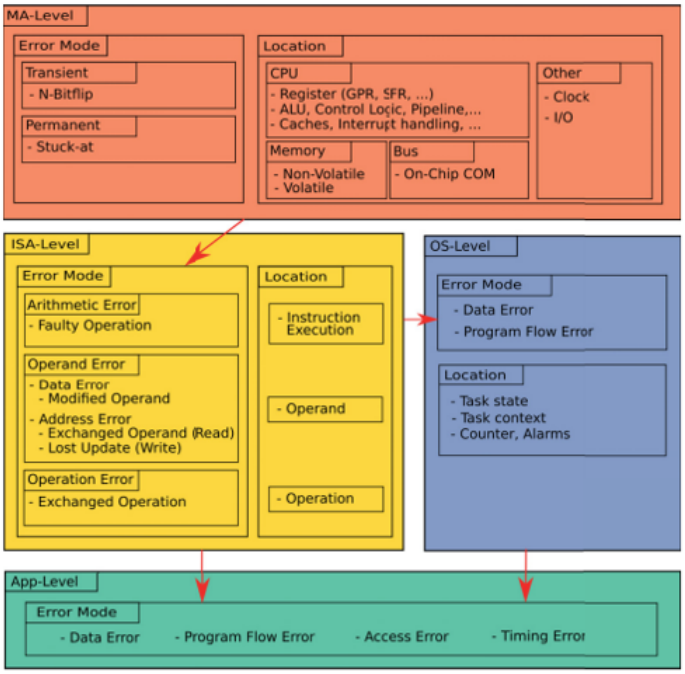

Figura 3.2. Abstraction levels of fault model

Faults could occur on various layers of a system: from the hardware layer down to the software layer. Examples of hardware faults might be bit flips, or multi-bit flips[19], while software faults are for design defects, timing errors, etc. As showed in fig 3.2, faults could propagate along a system. Similar to the operating system level, non-masked errors at the ISA level can propagate and manifest at the application level, in different ways. The main manifestations are data, program flow, access and timing errors [19]:

- *Data Errors:* represent erroneous values of parameters or variables in the system. Data errors can arise in a variety of ways, and can be

costly. Examples of data errors are data corruptions, data missing, inconsistencies, and so on;

- *Program Flow:* when in a software, the resulting behavior is different from that expected, such errors may occur. They possibly leading to missed, wrong or superfluous operations, being carried out due to an erroneous sequence or execution order;

- *Access Errors:* When accessing one of the system resources, such as a CPU, or a memory partition, access errors may occur. They could lead to a weakening of the applications isolation and affect the data of other applications. In fact, these errors could result in the generation of other types of errors, an invalid counter or pointer, and so on;

- *Timing Errors:* tasks are assigned by scheduling algorithms. When some of them are executed too early, too late or even not executed at all, timing errors can arise. In terms of scheduling, it's about deadline missing and dealing omission, so they appear at operating system level .Other factors that could generate these error type are changes in task priorities, omission of task activation and/or erroneous scheduling decisions;

3.2 Redundancy

Redundancy is a methodology aiming to improve the reliability and availability of a system. Generally speaking, electronic systems consisting of software and hardware components, in which redundancy can be achieved by applying extra copies of these components in parallel to handle the system workloads. Applying redundancy to a system involves increased complexity and additional costs. For this reason, it's better to limit it only in applications where the cost of failure is too high. For example, business-critical systems, safety-critical systems and systems that have a significant impact during downtime, belong to this category. In addition, many applications store sensitive information and data in databases. Therefore, for business continuity purposes, protecting databases with redundancy, must be a safety priority, in order to prevents catastrophic failures.

There are various methods, techniques, and terminologies for implementing redundancy and they depends on application type:

- *Parallel Redundancy*: having multiply units running in parallel, all units are highly synchronized and receive the same input information at the same time. Their output values are then compared and a voter decides which output values should be used. There are several topologies, such as Dual modular redundancy, Triple modular redundancy and Quadruple Modular Redundancy. This technique can be applied at both the software and hardware levels. The most widely used software redundancy techniques are N-version programming(NVP) and recovery block technique(RB);

- *Information redundancy*: such as error detection and correction methods;

- *Time redundancy*: performing the same operation multiple times such as multiple executions of a program or multiple copies of data transmitted. This method attempt to reduce the amount of additional hardware whereas time resource is not critical;

There are several redundancy structures and most used are classified into three macro categories:

- *Active redundancy:* when all redundant units in a system work concurrently. It can in turn be classified into 3 configurations: *Full*, *Partial* and *Conditional*. Full configuration works with all units activated with one surviving unit that ensures non-failure. In a partial configuration, only a minimum number of units can fail, to ensure a specific fault tolerance level. Majority voting systems often fall into this category; for example in a four units system, no more than two units can fail to achieve a majority. A conditional configuration is a form of redundancy that occurs according to the failure mode;

- *Standby redundancy*: involves backup auxiliary units. If failover occur, these units will take in place of the main units, in order to maintain a correct execution of the system;

- *Load sharing*: when a failure of one or more units, requires a workload distribution over the remaining active units;

- *Redundancy and repair*: where redundant units are subject to immediate or periodic repair. The reliability of a system depends on units reliability and by the repair times;

Other forms of multiple node redundancy structures are available that allow for greater redundancy and robust load balancing solutions and also possibility to realize hybrid solutions.

3.3 Software Redundancy

Software redundancy is an important problem taken into account by Software reliability engineering(SRE) [8] and it's a technique used for fault tolerance. Especially in the domain of safety-critical embedded and cyber-physical systems, is generally the optimal solution for increasing the reliability of these systems. Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [9]. In the literature, a number of techniques have been proposed to attack the software reliability engineering problems, based on software fault lifecycle:

- *Fault prevention:* trying to prevent failovers, by construction;
- *Fault removal:* using verification and validation methodologies to detect errors, in order to eliminate them;
- *Fault tolerance:* a unit of software is fault-tolerant if it can continue delivering the required service, even if an error occurred and not removed. These types of errors are called *software faults*. When these errors are generated, they produce errors in program flow, output errors, internal state, and so on. Software redundant techniques can be applied here to manage faults. In particular, main goal of this thesis is to propose software fault tolerance methodologies, applied to Docker container systems;
- *Fault/failure forecasting:* leveraging faults evaluation methodologies, it is possible to estimate when they might occur and what consequences might occur. Software reliability modeling focuses on these types of techniques;

Diversity: When redundancy is applied to a power or mechanical system, fall back strategies requiring the mere presence of another of the same type of component. The important difference between hardware redundancy and software systems redundancy is that the second one usually require extra configurations, on the host system. Specifically, when a software is redundant, all dormant software faults are also copied. Therefore, all N units in a fault

tolerance system, are designed and setted up, differently and independently. This is the concept of software design diversity. In other words, redundant units must to be different version of a specific unit. Thus, if anyone of the redundant version fails, at least one of the others will provide an acceptable output. The two basic models of fault software units are N-version software (NVS), shown in fig 3.3, and recovery blocks (RB), shown in fig 3.4.

3.3.1 NVP

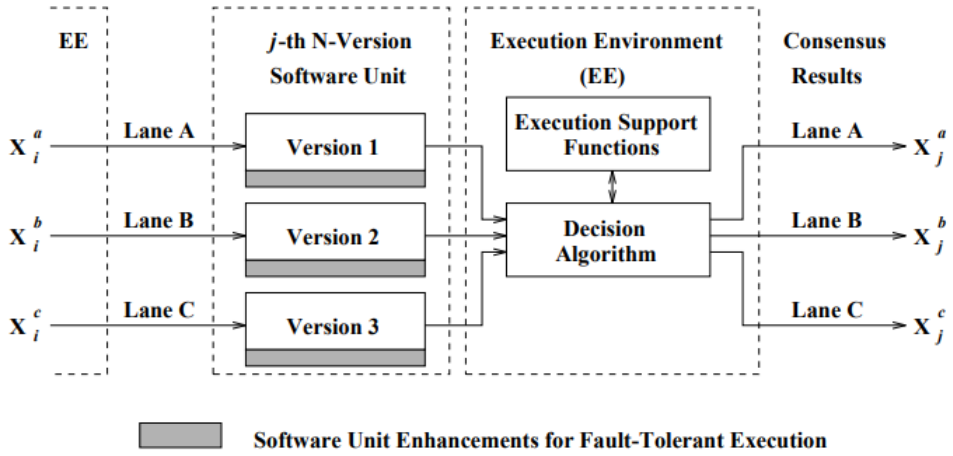

Figura 3.3. The N-version software (NVS) model with n = 3

N-version programming (NVP), also known as multi version programming, is a fault tolerant software that runs different version of the same algorithm. In other words, equivalent programs are independently generated from the same initial specifications. The units generate a series of outputs over time, and then they depends on a generic decision algorithm, also called voter, in order to determine a consensus result, comparing each output. If the units work perfectly, they will all generate same outputs, hence voter obtain unanimity. If one or more units fail, the voter processes the outputs, getting the majority.

An initial specification should define:

- which function should run each unit, belonging to an N-version fault tolerance system;
- data formats for the special mechanisms: comparison vectors (k-vectors), comparison status flags (cs-flags), and synchronization mechanisms;

- the cross-check points(cc-points) for c-vector generation;
- the comparison or voting algorithm, to be executed by voter unit;
- the response to the possible outcomes of matching or voting. The comparison algorithm explicitly states the allowable range of discrepancy in numerical results, if such a range exists;

As NVP is based on design diversity technique, the built program will fail independently and with low probability of coincidental failures. This ensures that one of the other versions will continue to provide the required functionality.

Probability of failures is directly proportional to complexity, in chip technologies. The need to introduce fault tolerance systems has increased over the years, due to the exponential growth of technologies. However, it's increasingly difficult to calculate the probability of fault, being that a complete design verification, especially in VLSI circuits, is very hard to achieve.

Using N-versions of a software, allows to a system to continue operations, even if faults are occurred. Furthermore, Software verification and validation time is reduced, executing two independent versions in an operational environment. In this way, complete verification and validation, with concurrent production operation, is achieved.

A huge advantage of NVP is that it allows programmers working with their own time and location. In fact, given a formal and an effective specification, different versions of software can be written by different programmers, using their own personal computing equipment. Expecially in highly controlled professional programming environments, this approach will drastically bring down the cost of programming.

3.3.2 Recovery Block Technique

The system consists of using a series of modules, called recovery blocks, which are executed one at a time. The process begin by starting the primary module. An acceptability test is performed on this module: if the acceptance test determines that the output of the primary module is not acceptable, it recovers or rolls back the state of the system before the primary module is executed. Therefore, the primary module is deactivated and the secondary module takes its place. The process, carried out on the first module, is repeated on the second module as well. Thus, if the acceptance test is failed, a third module will be activated, and so on. When all alternate modules

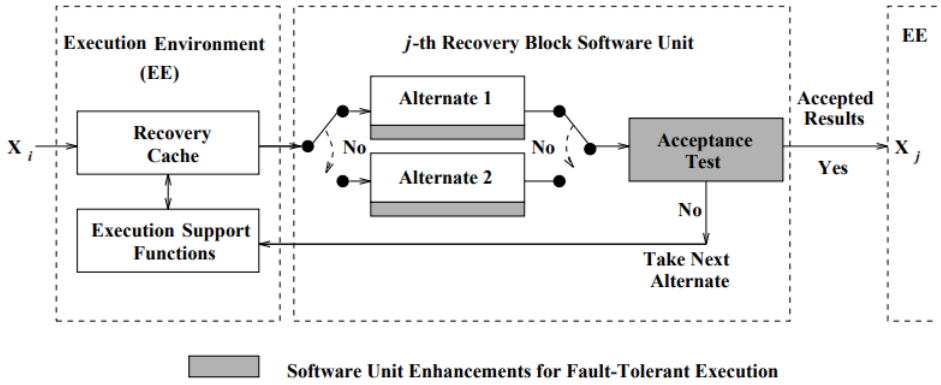

Figura 3.4. The recovery block (RB) model

are exhausted, the recovery block itself is considered to have failed and the entire system goes down. In order to minimizes the total system cost, given the reliability of each module, reliability optimization models has been designed. These models allow to determine the optimal number of modules, to introduce in an RB system.

In a N-version programming, all modules are executed in parallel, while modules are execute sequentially in a recovery block system. Finally, The recovery block, generally, is not the best solution in critical systems where, one of the requirements is to have an optimized real-time response.

Capitolo 4

Container Classes

As discussed in the previous chapter, redundancy is necessary to guarantee software reliability. Aim of this work is to illustrate methodologies to introduce software redundancy through Docker container technology. The main idea is to instantiate a desired number of containers, which will identify the level of redundancy. Therefore, this fault-tolerance system will require the installation of additional modules that allow monitoring of specific characteristics and outputs of all containers. Results obtained from this monitoring are analyzed by voting blocks and ensure correct outputs.

Because containers have different characteristics and features, they were organized into container classes:

- *Web server class;*
- *Database class;*
- *Monitoring class;*
- *Programming application class;*
- *Continuous Integration class;*
- *Service Discovery class;*

4.1 Docker compose parsing

Compose is a tool for defining and running multi-container Docker applications. It's possible to define and configure all application's services, by

compiling a file in YAML format. Then, with a single command, all the services are created and started, depending on these configurations.

First step is to write a compose file, observing composition rules determined by Docker. In order to implement a fault tolerant system, another version of this file must be properly generated. This task is achieved by writing additional sections, for each container which needs redundancy. All container classes, define additional entries to introduce in a standard compose file, which have different values and/or types. Therefore, a computational block needs to be properly designed to performs:

- Parsing operation, by searching all required parameters for fault-tolerant containers;
- Knowing in advance which additional containers are needed to perform voting operations (for instance the voter), it place them in the new compose. These containers are defined in different Dockerfiles, previously prepared;

Parser block leverages Docker feature of passing parameters through Environment variables. Containers can be setted up in this way, therefore voters are parametrized to manage operations. Voter architectures depends on container classes because, for instance, operations to execute are more or less complex or input formats from container system can be different.

This approach create a subsystem level that doesn't need to manage, thanks to the automatic deployment. Therefore, Docker managers only need to specify fault tolerant parameters, without having to worry about deploying of additional containers.

4.2 Fault tolerance parameters

Fault-tolerant entries for each container class are mainly four: *input*, *output*, *storage* and $ft\_parameter$. All entries have different value or type, depending on the class:

- *input:* this entry define format and type of input to be sent to all redundant containers;
- *outputs:* main goal of this work is to identify outputs coming out from each container class, which then have to be observed by the voting system;

- *storage:* each container class could manage files and directories in different ways. They needs to be accessible, shared among fault-tolerant system and configured, if necessary, to perform computations;
- *ft_parameter:* sets the level of redundancy, which blocks to instantiate, how to manage resources between containers or properly configurations for containers;

4.3 Database class

```
########################
#### DATABASE CLASS ####
########################
inputs:
    queries: shell/webserver
outputs:
    databases: True/False
    Tables: True/False
    logs:
        enabled: True/False
        redirection: single file/multiple files/stream
        format: SQL statements
storage:
    database: 'directories'
    logs: 'directories'
    config: 'directories'
ft_parameters:
    ft_level: N
    forwarding: 'block name'
```

Parser block starts deploying number of redundant database containers, reading fault-tolerant parameter, named *ft_level*. After that, is always mandatory to deploy a container that send the same inputs to the redundant container system, reading *input* entries. In this way, it is possible to make the voting of containers outputs. This block is specified in the *forwarding* entry. The approach is to prepare a Dockerfile with a precompiled container that perform this tunneling work or to use a container builded on an image from DockerHub.

A database class container is characterized by data exposure based on a system of folders and subfolders. Voter container performs control operations

on files, such as databases or tables. Therefore, in the respective entries, it's possible to activate the voting on these files and specify the level of comparison to be made, providing it with parameters in the form of environment variables or files.

All data to be checked must be specified in the entry *storage*, letting Docker to create all Docker volumes needed. Volume is a useful and easy Docker feature, used to share these data among the voter and containers.

Fault tolerant system strategy, works by doing following operations:

- Whenever a new query incoming in input, tunneling container split the same query among all fault tolerant containers. In addition, a trigger is sended to the voter, in order to perform its operations;

- The voter read last log, as specified in the respective entry, to identify query type;

- Depending on log reading, it chooses properly operations and votes outputs. If something goes wrong, it notify errors, and conversely it provides correct outputs;

4.3.1 Database Logs

As showed in the fault tolerant specifications, logs have three entries, which affects voter behavior. A database class has generally disabled logging into files, by default. Consequently, it is necessary to be able to activate them in the fault tolerance system, by acting on the respective environment variables. Alternatively, it may acts on the configuration files, but in this way, relative directories must be specified inside the $config$ entry. Redirection into file or files is specified and, with it, related log directory in the *storage* section. Log $format$ depends on database type, so voter needs to know how logs will be readed.

The voter must be sure that the last log readed, related to the same incoming query, matches for all containers. This ensures that the voting operation is carried out on the correct outputs. So, voter must also perform this log comparation. However, logs may contain variable informations, like time stamps. As a result, may not match, even if the query is the same for all containers. Voter must take into account all random informations, before making operations. Random variables could also occur in other types of data, voter exploit byte contents checking only the relevant ones.

4.4 Web Server Class

```
##########################
#### WEB SERVER CLASS ####
##########################
inputs:
    requests: HTTP/HTTPS
    domains:
outputs:
    html: True/False
    files: JSON/image/audio/video/ecc
    logs:
        enabled: True/False
        redirection: stream
        buffered: True/False
        format: Web
storage:
    html: 'directory'
    data: 'directory'
    configs: 'directory'
    logs: '../<container id>-json.log'
ft_parameters:
    ft_level: N
    forwarding: 'block name'
        configs:
        labels:
        ...
    routing_level: M
    load_balancing:
        enabled: True/False
        algorithm: Round Robin/Least-connected/..
```

Web servers are programs that make website files and programs accessible to web browsers over a network. They are running software that receive and elaborate HTTP requests, which is the protocol that browsers (or other type of clients, such as software applications) use to view webpages. They can be accessible through domain names of the websites that store, and thy delivers the content of these hosted websites to the end user's devices. Whenever a browser needs files hosted on a web server, it requests them via HTTP protocol. When the request reaches the correct (hardware) web server, the (software) HTTP server accepts the request, finds the requested documents,

and sends it back to the browser, also through HTTP. If the server doesn't find the requested document, it returns a 404 response instead. For this reason, in the entry *input*, HTTP requests have to handled by fault-tolerant system.

As done in each other classes, the *ft_level* parameter is readed, to determine how many redundant containers will be deployed. The container deployment, in this case, that performs forwarding of HTTP/HTTPS requests, can be a reverse proxy. It differs, for example, from the database class, whose inputs consist of different formats. Adopting a reverse proxy for forwarding work allows to take advantage of its features, such as load balancing, uses of middlewares, encryption supports, etc. If the reverse proxy allows it, it could be useful to modify the load balancing algorithm, by acting on *algorithm* entry.

4.4.1 Load Balancing Problem

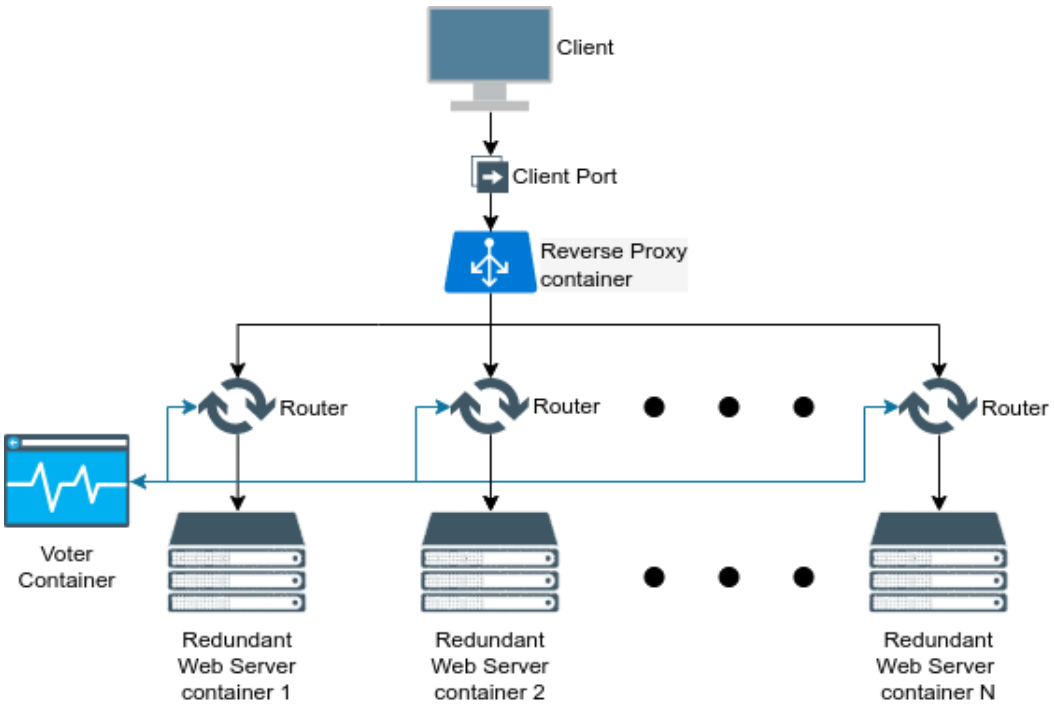

Figura 4.1. Web server class redundant scheme, without Load Balancing

An important feature of a reverse proxy to consider is load balancing. It is an effective mechanism for managing container failovers. It increase application

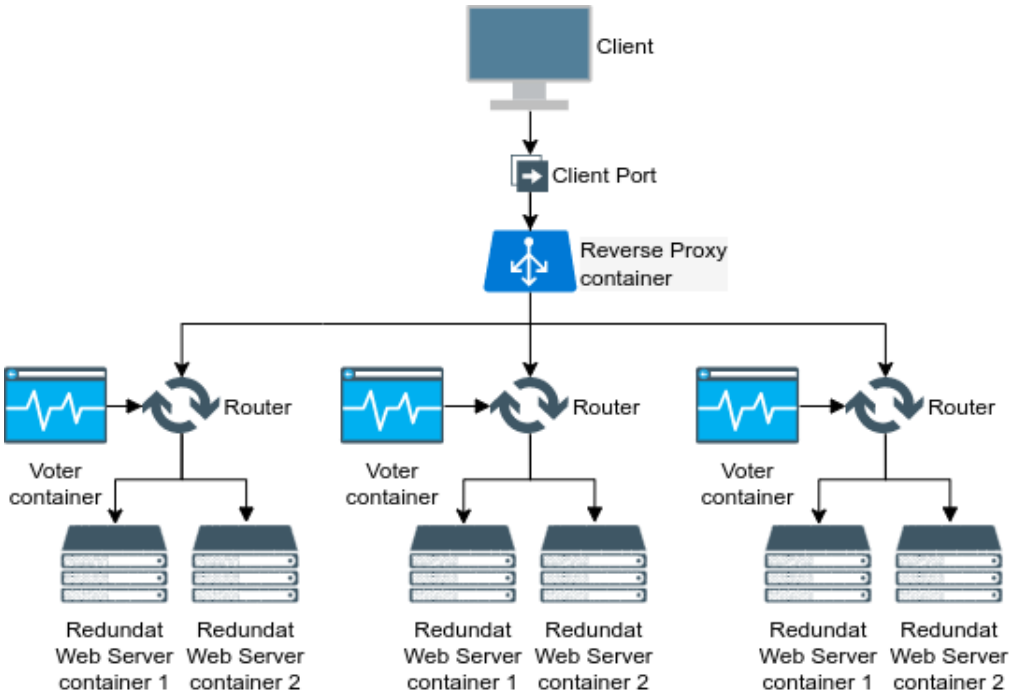

Figura 4.2. Web server class scheme with Load Balancing ($ft\_level = 2$, $routing\_level = 3$)

availability and performance by distributing traffic across more than one server. To ensure requests are assigned to servers, that can handle the traffic, many load balancers monitor server health and implement failover execptions. Health check monitor, periodically sends HTTP/HTTPS requests to server pools to monitor their status. If the HTTP/HTTPS check reveals that a server is unhealthy or offline, load balancer will reroute traffic to an available server.

A container introduced into the fault-tolerance system, may have a load balancer enabled. The information regarding load balancing is transmitted to the parser via the parameter: *load_balancing*. It is essential to take this feature into account, as it may compromise the voting mechanism.

Two types of methodologies can be adopted: one involving the load balancer activated, and one without load balancer. As an example, the methodology used in this container class without load balancer, is shown in fig 4.1. This approach works exactly like the database class, with the only difference being the introduction of routing blocks:

- The parser instantiates a number of containers equal to the parameter:

ft_level;

- HTTP requests are forwarded simultaneously to the routing blocks, which in turn forward them to redundant web servers;
- Requests processed by web servers are sended back to routers, which forward the content to voter for processing;
- Voters know the request type by reading the last log, which must be the same from all containers. Next, it votes the outputs from the web servers and returns a correct copy to the routers;

Routers handle data traffic that flows among redundant web servers, reverse proxy and the voter container. Furthermore, they allow a correct outputs synchronization between web servers and the voter.

The methodology that leverages use of the load balancer is more complex. In this case, the parameter *routing_level* must also be specified. The parameter ft_level, in this case, assigns a number N of redundant web servers behind each router, while *routing_level* defines number of routers connected to the reverse proxy. The reverse proxy redirects traffic to one of the connected routers, therefore 1 voter must be assigned for each path. The difference with the first solution is that is more resource-intensive, but it increase reliability of the fault tolerant system. Furthermore, it fits well in cluster systems, where each path can be assigned to a different node, maintaining the same level of fault-tolerance on each node.

4.4.2 Web server logs

Most of the logs generated by web servers, are redirected to stdout and stderr. This feature is marked in the $redirection = stream$, inside the *logs* entry. In this configuration, Docker handles this stream by default, saving logs output to the local host, in JSON format. Therefore it's possible to access these logs, from outside of the container, pointing to the local directories, in the *entry* storage. Docker organizes directories by container IDs, so they must be communicated to the voting system, during the deployment phase.

Another problem is the log buffering. In this configuration, a log buffer is always open, and a web server continuously write data on it. Whenever is full, container deploy all the content to stdout, and finally Docker can print the logs. To overcome this issue, a configuration file inside containers has to be deployed, in order to set the disable option. To avoid writing a configuration

file, some containers allow management of environment variables, that can be setted up during the development phase.

4.5 Programming Application class

```
########################################
#### PROGRAMMING APPLICATION CLASS ####
########################################
inputs:
    codes: C/Python/..
outputs:
    files:
        format: audio/video/...
        stream: serialized/parallel
        type: short process/ long running
    stream:
        enabled: True/False
        redirection: files/None
        buffered: True/False
    logs:
        enabled: True/False
        redirection: stream
        buffered: True/False
        format: custom
storage:
    entrypoints: 'directories'
ft_parameters:
    ft_level: N
    maximum_cores: M
```

Simplest case is an application software running on a single host, i.e. a computer program that performs a specific function. They are designed to facilitate a large number of functions, such as: data and information management, visuals and video development, word processing, web browsers, graphics and so on. Software applications are classified in respect of the programming language in which the source code is written or executed, and respect of their purpose and outputs.

Simplest and cheapest approach, in order to apply redundant mechanism, is to carry out *parsing operation and Dockerfile preparation*. Dockerization of

an application starts preparing the relative Dockerfile, that specify following parameters:

- *Choose a base Image:* Docker build filesystems layers starting on the base image that is the read-only layer used to run containers. It could be mounted from the Docker Hub or from a custom image;
- *Install the necessary packages:* Depending on the base image, addition packages could be necessary in order to run custom application correctly;
- *Add custom files:* using ADD or/and COPY command, interpreted applications (PHP, Python, C, etc.) have to be added, paying attention to folders and permissions to assign. Using an application specific configuration file, it's also possible to define format, fields, location, environment variables and so on;

In a multiple containers approach, docker-compose file needs to be prepared. All containers run executable files defined in Dockerfile and the better way is to define a docker entrypoint script where all computational outputs are redirected to specific directories. As done with other classes, *parsing* of the compose file is performed, in order to write another compose-file with redundant containers for all containerized applications. Furthermore, voter container must be generated, in order to perform comparisons between outputs coming from original containers and their redundant counterpart.

Voter can be designed with different fault tolerance structure, as discussed in the previous chapter, and all redundant containers must introduce diversity to garantuee redundant mechanism work well. Furthermore, parsing have to take into account *outputs format* in order to set operations properly, being that comparisons could be diversified. This task is accomplished inserting the format, in the entry $files$, within fault tolerant section. In this way, parser can generate correct voters, reading fault tolerance parameters attached to containers.

Another possibility is to enable *byte-to-byte reading* in the voters. This methodology, allow to determine output format by reading output files. This makes voters as generalized as possible, but this approach increase voting complexity. Several cases must be take into account to avoid misoperations:

Ouputs sharing: outputs sharing is possible by defining volumes. Docker volumes are file systems mounted on Docker containers to preserve data generated by the running containers. Volumes are stored on the host, independent of the container life cycle. This allows users to back up data and

share file systems between containers easily. Parser have to duplicate volumes for all redundant containers to avoid race condition. A single container could share many volumes, so parser have to duplicate all of them for the redundant counterpart.

Outputs could be shared by other sharing mechanism, for instance by sockets. In this case, docker-compose must specify exposed ports for containers. For standalone containers, it's possible to remove network isolation among all containers and the host, using the host's networking directly. It is important that the voter receives outputs on different ports for each container, at the cost of too many busy ports.

Another option is to share ports but syncronization mechanisms are necessary between communication among several containers. Docker offer driver networking, such as the *bridge network*. It's a link layer which forwards traffic between network segments. This bridge is software device running within a host machine's kernel and allows containers, connected to the same bridge network, to communicate each other. Containers which are not connected to the same bridge network, cannot communicate directly. Network communication is useful whenever outputs format, generated by containers, are not files. This is because IPC mechanisms can be used to transfer data. In order to reduce complication, after parsing of the compose-file, output containers could be redirected into a file as entrypoints.

CPU assignment: By default, each container's access to the host machine's CPU cycles is unlimited. It's possible to set various constraints to limit a given container's access to the host machine's CPU cycles. Most users use and configure the default CFS scheduler or the real-time scheduler. Best choice is to assign different cores among original containers, redundant containers and voters, specifying *maximum_cores* in the fault tolerance.

Syncronization: Application can generate multiple outputs sequentially and in different time intervals. Moreover, they couldn't be syncronized with the redundant ones, therefore voters needs to manage syncronization between outputs. They can afford it by checking outputs whenever a new one is created and if exist the relative redundant outputs. Thus, it's freezed when containers aren't syncronized and waiting for valid outputs.

Whenever outputs are printed to stdout, users must be redirect printed output into a files, in order to share among containers. Moreover, Operating system cannot manage delay between output generation and redirection, thus

there is out of syncronization, also in this case. This solution can afford syncronization, even without operating system working, because syncronization management are left to voters.

Logging: it's not necessary to get all deeply informations from all containers, in this kind of environments, [28]. Tuning and troubleshooting can be performed in real time if needed and standard informations, such as RAM and CPU consuming, are enough to proper management. After parsing of the compose file with redundant containers, it could be istantiate another container that receive and elaborate logs from entire containerized system. Optimal choices are external services like: cAdvisor, Sysdig and so on.

Advantages of parsing and Dockerfile preparation approach are: easy management, easy preparation of the system, customizable voters and fixed resource assignment. However, it's a static approach, thus Docker limitations cannot be avoided. Whenever applications require fined tuning, advanced methodologies like MDE, as showed in chapter 4.8, can be applied.

4.6 Monitoring class containers

```
##########################
#### MONITORING CLASS ####
##########################
inputs:
    metrics:
        Counter: True/False
        Gauges: True/False
        Histograms: True/False
        Summaries: True/False
        ...
        format: Time series/Event data/Real-time data/..
    query: HTTP/ DQL
outputs:
    data points: Time series/Event data/Real-time data/..
    data sources:
        type: 'monitor_name'
        access: 'proxy'
        org_id: 1
        url: 'http://monitor_name:9090'
```

```
            is_default: true
            version: 1
            editable: true
    storage:
        provisioning: 'directory'
        config: 'directory'
        datasource: 'directory'
        database: 'directory'
    ft_parameters:
        ft_level: N
        forwarding: 'block name'
            scraping_interval: N sec
        alert manager:
            name: Alertmanager
            type: alertmanager
            url: http://localhost:9093
            access: proxy
            #optionally
            #basicAuth: true
            #basicAuthUser: my_user
            #basicAuthPassword: test_password
```

This class of containers includes monitoring softwares. When a system or an entire infrastructure are administrated, it is necessary to ensure that the different system element services are running smoothly, in order to keep services going as expected. Moreover, they rely on metrics to monitor and understand the performance of their applications and infrastructure. System monitoring containers (such as cAdvisor, Syslog, Prometheus) helps in resolving those issues, which may lead to a significant break in the systems.

A possible methodology to structure a fault tolerance system for this class of container is the adoption of a data analyzer and monitoring tool, works as a voter block. The parser distributes, as usual, a number of monitor containers equal to ft_level and forwards the same inputs to them.

The data analyzer picks up metrics processed by the monitoring containers, which can be leveraged to perform voting. Voting is achieved by loading custom dashboards into the analyzer. A dashboard is a set of one or more panels, organized and arranged into one or more rows. Each of them can be configured to perform custom operations on the controlled metrics. These tools have alert manager built-in softwares, which allow to notify and handling certain events. In this fault-tolerant system, it's used to notify voting

status: whenever an error voting occur, a notification is sended by alert manager.

4.6.1 Inputs parameter

This class receives input types defined in the respective entry, which are generally of two types: *metrics* and *queries*. A monitor starts identifying one or more targets, called endpoints, and then retrieving metrics. Metrics represent the data in a system i.e., in other words, numeric measurements. What users want to measure differs from application to application. For example, for a web server it might be request times, for a database it might be number of active connections or number of active queries, and so on. A container monitor stores all scraped samples locally, so directories have to be defined in *storage* entry. Moreover, all collected data are saved in a specific format, as specified in $format$ entry.

Metrics are retrieved via HTTP requests or by query language, so an entry called *query* taking into account this inputs.

Since the inputs can be either files or streams, the forwarding block must be programmed to handle them, simultaneously. Metrics management can be handled differently from other classes, by leveraging Docker volumes. Since monitoring containers process metrics by working on specific working directories, inputs can be shared simply by using Docker volumes. The goal of the forwarding block is to take metrics from target containers and move them to local volumes, accessible among all monitor containers. This operation "masks" this volumes from the external service. However, it must take into account the scraping interval, for all monitors to pick up the data only after the block has moved them into the volumes.

Furthermore, it needs to forward external queries to all monitors, and then send back metrics. In this case, there is no need for a trigger mechanism, the data analyzer container is already synchronized with the generation of all voter outputs, thanks by proper dashboard configurations.

4.6.2 Data analyzer configurations

In order to set up the system, the parser must retrieve the informations from adding fault tolerance parameters. In the *datasources*, inside the *output* parameter, sources of the metrics to be voted by the data analyzer must be specified, and also informations about connections, ports, and so on. Sources are all redundant monitoring containers. This allows to create necessary

configuration files to allocate inside the container. Also in this case, parser known locations of this files looking at the *storage* parameter.

Data analyzers allow the configuration of alert managers, which in this case is specified in the general fault tolerance parameters.

4.7 Continuous Integration class

```
    ########################################
    #### CONTINUOUS INTEGRATION CLASS ####
    ########################################
    inputs:
        sources: GitHub/local/...
        pipeline:
            source: Jenkinsfile
            format: declarative/scripted
        signals: HUP/INT/KILL/TERM/...
    outputs:
        automations: single task/pipeline/multibranch pipe
        build stage:
            source: Dockerfile/...
        test stage:
            code: Python/C/..
            results: console/junit/..
        deployment stage:
            results: console/file/...
            sources: GitHub/local/..
        logs:
            enabled: True/False
            redirection: Log file/stream
            buffered: True/False
            format: standard
        external_logs:
            enabled: True/False
            redirection: Log file/stream
            buffered: True/False
            format: standard/custom/...
    storage:
        repository: 'URL/directory'
        Workspace: 'directory'
        test: 'directory'
```

```
        logs: 'directory'
        external_logs: 'directory'
    ft_parameters:
        ft_level: N
        forwarding: 'block name'
```

Continuous Integration is a automation process that allow cooperation and many changes, between different contributors, into a unified project. The process consists of several steps, called stages. A stage block defines a certain number of tasks, to be executed along the entire process:

1. after the code commission, continuous integration process starts;
2. building and testing of the code;
3. if testing is successfully, building is ready for deployment;
4. pushing into production,

First outputs to be voted on, in this container class, are builded codes. Therefore, in the *buildstage* parameter, it has to be specified the type of file or process that the voter should know about. Accordingly, the parser instantiates a Docker volume containing the building directory, accessible by the voter, specified in the *storage* > *workspace* parameter. For example, using Jenkins, the building directory is often: "*Jenkins/jobs/project_name/builds/build_id*".

In the testing phase, builded code is executed and its outputs have to be voted. Such code, may be written in different programming languages, so it must be specified, in the *teststage* > *code* entry. By knowing which programming language needs to be voted on, voting algorithms take into account a specific syntax, how mapping memory is performed, functions, and so on. Continuous integration software performs testing on the executed code, printing the results in a specific file format or stream, which can be viewed in the *results* entry. Voting results, should be compared with the results coming from the CI software, in order to ensure a correct execution of the testing stage.

Deployment stage could consist of different operations, such as pushing code into a production system or in a simple check that previous stages are completed successfully. Therefore , it is only necessary to know the directories or output stream.

After you run a job, Jenkins gives you access to log data. You will find this extremely important to figure out why things failed and how you can fix things for future runs.

For each stage of execution, logs are generated. Monitoring these logs allows voting to be managed appropriately. Some software for continuous integrations, allow the use of customizable logs. In that case, they have to be specified all the necessary parameters, in order to send them to the voter.

4.8 Service Discovery Class

```
##################################
#### SERVICE DISCOVERY CLASS ####
##################################
inputs:
    queries: HTTP API/DNS API
    commands: SIGHUP
outputs:
    discovery:
        results: stream/txt
        format: SRV
    reload: file
    healthchecks:
        source: script
        timeout: 30s
        enable: True/False
        type: TCP connection/Docker API/...
    kv:
        source: file
        restriction: True/False
    logs:
        enabled: True/False
        redirection: Log File/stream
        buffered: True/False
        format: standard
storage:
    config: 'directory'
    data: 'directory'
ft_parameters:
    ft_level: N
    forwarding: 'block name'
    masked_IP: xxx.xxx.xxx.xxx
```

Classical architecture, for deliver an application, is called *Monolithic*. Applications are packaged and deployed as a single unit, having a specific language

and framework. Monolithic architectures have several advantages: easy deployment, easy testing, and easy deployment. However, main drawbacks are:

- *Low reliability*: if there is a bug in one of the submodules, the entire application needs to be reployed;
- *Complexity*: Size is limited to avoid slowing down performance too much;
- *Hard continuous deployment:* different modules have conflicting resource requirements, so it is difficult to make changes and updates;

Microservices architecture is the answer for overcoming the limitations of monolithic architecture. The idea is to split applications into a set of smaller and interconnected units, which communicate through a well-defined, lightweight mechanism, to perform tasks. Service mesh tools, for connecting and configuring distributed infrastructures, are needed to manage these types of architectures. The main features of these tools are:

- *Service Mesh*: this service enables inter-communications between different services, in a distributed application. Applications can use sidecar proxies, in a service mesh configuration, in order to establish TLS connection and to allow communications without modifying components;
- *Service Discovery*: is a registry, saved in a database, where all network locations of services are saved, and made discoverable via a DNS or HTTP interface;
- *Health Checking*: issues and failovers could occur in a cluster system, so some tools offer health checking mechanisms, to handle these events. Proper services are activated, avoiding a shutdown of an entire system, by preventing all traffic sended to malfunctioning nodes;
- *Key/value Storage*: is a storage type that offers several advantages, such as leader election feature, dynamic configurations, flagging, and so on;

In order to properly deploy a redundant system, for this container class, it is necessary to manage its client-server architecture. As an example, Consul service is shown in fig 4.3, which represent a typical architecture belonging to this fault tolerance class. It can be seen that service discovery container works in a cluster, where each instance can work in both client or server mode.

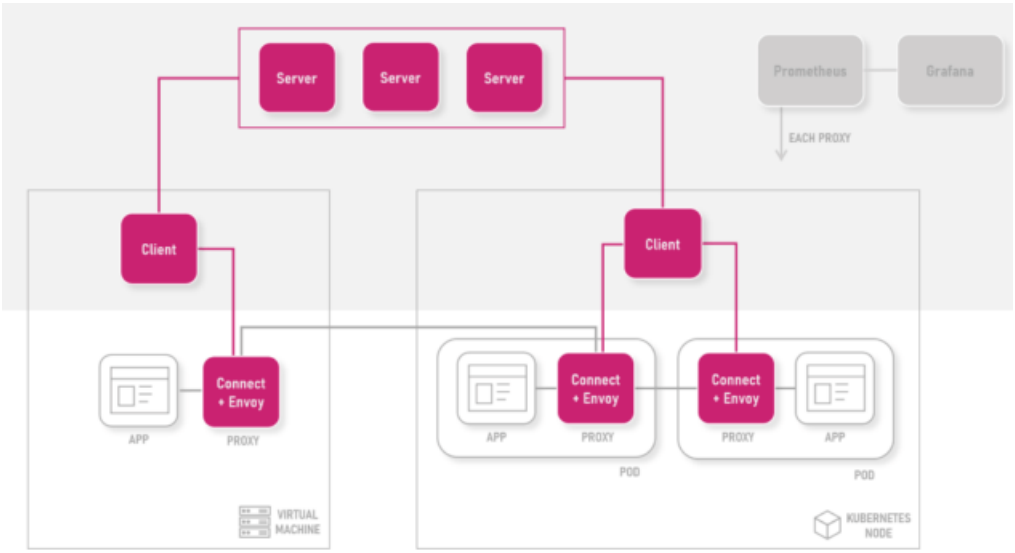

Figura 4.3. Service discovery class typical architecture

Therefore, in order to enable proper cluster management, it is necessary to specify a forwarding block $masked\_IP$, which will be deployed in front of the redundant containers. In this way, all containers belonging to the cluster, will refer to this IP address, regardless of the redundancy level chosen for that specific container.

A redundancy system, deployed for this container class, must be able to proper handle each feature services. In general, cluster configurations are saved in a database. When an update needs to be made to this type of file, trigger commands are sent, in order to restart that container with new configurations. Therefore, a redundancy system, has to handle this input type, as shown in $input > commands$, observe how reload operation is performed, via *reload* parameter, and find storage locations.

One of the major use cases for this container class is the service discovery. Services can find their dependencies along the cluster, leveraging appropriate DNS or HTTP interfaces, in order to access the discovery registry. Each time this task is performed, the voter must check the outputs generated by redundant containers, as specified in the $output > discovery$ parameter. Furthermore, whenever the discovery registry is updated with a different service, the voter must ensure successful connection with the system. In addition, logs are generated for each access, so they must be compared to associate tasks with checks.

If this class provides health checking algorithms, they should be voted on. They are generally external scripts that are executed, so it's necessary to determine specifications of these scripts, in the *healthchecks* parameter.

Finally, key value datastore, is distributed on containers, and can be accessed by any agent, both in client and server mode. Many architectures are designed to replicate data automatically, across all containers connected. This type of storage is used to deploy configurations and metadata, across all services belonging to a microservice architecture, dynamically. Having a quorum of servers will decrease the risk of data loss if an outage occurs. Natively, containers forward requests to servers, including key/value reads and writes. Whenever, this type of access is made, the voter checks storage endpoints and compares the contents. Therefore, it is important that access synchronism is maintained between redundant containers.

4.9 Model-driven Redundancy

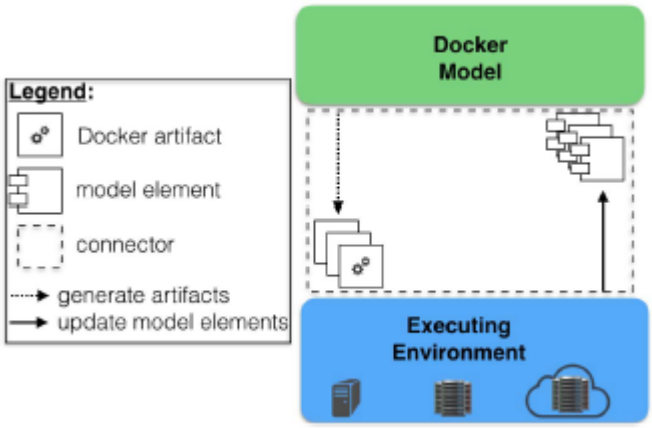

Figura 4.4. Model-driven Docker system architecture

As showed in the first chapter, container technology provides several advantages: it's a lightweight, portable and low-consumption technology. However, Docker has relevant problems, as argued in many applications [26], [27]:

- *Lack of verification*: once designed, the deployment of the containers have to be verified, because different problem could occurred such as misconfigurations, communication between containers, insufficient resources provided by the hosts to containers, human errors and so on. Docker

doesn't perform verification that deployed containers are conform with those designed;

- *Resources management at runtime*: Docker gives the possibility to set the resources, such as memory, CPU usage, disk, network and so on, only at design time. This is a Docker limit because workload could change run-time and containers resources should increase or decrease, if it's necessary;

- *Synchronization between deployed and executing containers*: Modification can be done in the design, such as adding new containers, changes configurations or links in existing containers, ecc. Updates require a restart of the environment because Docker doesn't provide synchronization mechanisms;

Attractive and very efficient solution to face off these limitations, is *Model-driven engineering (MDE)*. MDE is a methodology aims to develop software leveraging models, metamodels and model transformation techniques, which help the specification of translations between different model types.

These technique could be used to design abstractions of a running system, by developing run-time models, from different problem space perspectives. Thus, a run-time model can target a certain self-management capability and dynamically perform required operations.

The Eclipse Modeling Framework (EMF) is the main technology in Eclipse software for model-driven engineering. EMF is a modelling and a code generator framework for building tools and other applications based on a structured data model. This model is written in *Ecore* language and EMF generate *Java entities*, starting from this model. They are based on java API composed by several setter and getter, which store data of the application or tool modelled.

Starting architecture of this approach is to depict the entire system in three parts[26], as showed in fig 4.4:

- *Docker Model*: represent entire containerized system, which elements contains all container properties;

- *Docker Connector*: is a tool which interacts directly with the Docker daemon though HTTP, using a Docker API. It performs changes in the environment based on information received by Docker Daemon. It's possible for instance a generation of Docker commands, Docker Compose file, Docker Swarm configurations, modify container resource managing cgroups and so on;

- *Executing Environment*: it represents container deployed in execution in the cloud infrastructure;

Docker connector is implemented as Eclipse plug-in, and receiving run-time informations from Docker daemon, can fix the three problems illustated before.

Key concept of model-driven redundancy is to implement another Eclipse plug-in that receive redundancy properties, chosen by the user. After selecting redundancy properties at the design time, Docker connector performs artifact instantiations, such as voters and redundant containers and handle syncronization between container outputs and artifacts generation. Leveraging interaction with Docker daemon, user only needs to interact with a redundancy UI and letting connector the management. Interface properties, such as level-k redundancy, redundancy type, which components needs to be redundant and so on, have to be selected in order to depict software class. Classes provide redundancy tasks to connector depending on the applications, requirement and costs.

Capitolo 5

Methodology Validation

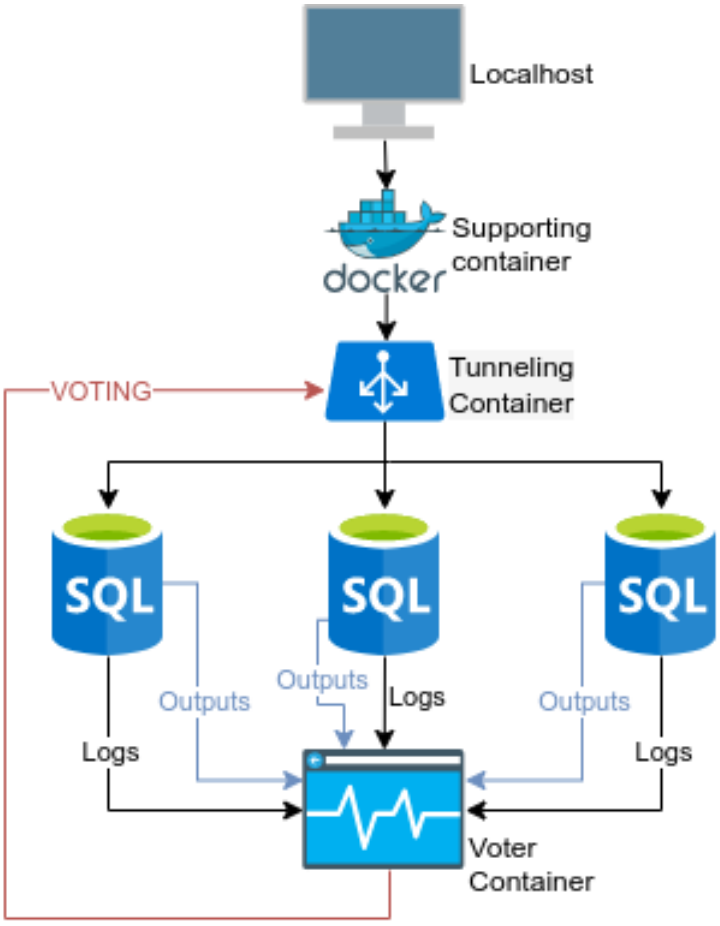

Figura 5.1. MySQL databases fault tolerance system

In the previous chapter, has been explained how containers can be classified, for instantiate a containers fault tolerance system. The objective of this chapter is to show how a methodology can be applied and to test its validity. Therefore, a *MySQL* database container is chosen, which represents a container class belonging to "*Database container class*".

MySQL is the world's most popular open source relational database management system. With its proven performance, reliability and ease-of-use, MySQL has become the leading database choice for a wide range of applications, websites and services.

To demonstrate the methodology, it has been deployed a mySQL container, on which fault tolerance has been applied, and a supporting container, just to access to the database, through port 3306. Port 3306 is the default port for the classic MySQL protocol, which is used by mysql clients, MySQL Connectors, other softwares, or other utilities. The fault tolerance system obtained consists of the deployment of 3 redundant MySQL databases, a voter container and a tunneling container, as showed in the fig 5.1. Queries coming from the supporting container are forwarded to each MySQL database, and a trigger signal activates the voting container. The voter reads the last last log generated by each database, in order to know the type of query, and then performs the voting operations appropriately. The result of the voting is sent to the tunneling container, which returns back the correct output selected by the voter.

5.1 Dockerfile Preparation

Docker builds images automatically by reading the instructions from a Dockerfile, which is a text file that contains all commands needed to build a given image. The initial Dockerfile deploy a MySQL container, builded upon the official image from DockerHub, which is the official repository of container images, provided by Docker:

```
### Starting Dockerfile ###
version: '3.9'

services:
    db1:
        image: mysql
        ports:
            - "3306"
```

```
            environment:
                MYSQL_ROOT_PASSWORD: admin

        sup:
            image: supporting_image
            build:
                context: .
                dockerfile: Dockerfile_supp
            container_name: supp_container
            hostname: 'supporting'
            ports:
                - "3306"
```

A system of three redundant databases, implies a fault tolerance level equal to three, i.e. $ft\_level = 3$. As a result, the fault tolerance section has been written as follows:

```
    ### Starting Dockerfile ###
    version: '3.9'

    services:
        db1:
            image: mysql
            ports:
                - "3306"
            environment:
                MYSQL_ROOT_PASSWORD: admin
            ### Fault tolerance section ###
            inputs:
                queries: shell
            outputs:
                databases: True
                tables: True
            logs:
                enabled: True
                redirection: single file
                format: SQL_statements
            storage:
                database: '/var/lib/mysql'
                logs: '/var/lib/mysql'
                config: '/etc/mysql/conf.d'
            ft_parameters:
```

```
            ft_level: 3
            forwarding: "tunneling"
        ##############################

    sup:
        image: supporting_image
        build:
            context: .
            dockerfile: Dockerfile_supp
        container_name: supp_container
        hostname: 'supporting'
        ports:
            - "3306"
```

Input type are queries, so a tunneling block will be deployed that forward this traffic to all databases through socket connected to port 3306, preparing a Dockerfile and setting $forwarding = tunneling$ name in the entry. The port information is passed from the parser to the tunneling container, reading the compose file. It will be builded a custom image, for a tunneling container and the voter container deployment. In the *storage* parameter are present all needed directories:

- *databases: '/var/lib/mysql':* MySQL has a unique directory, which contains databases. This entry allows to create a shared volume among all containers. This volume will be instantiated, only if database voting is enabled, checking the entry: $output > databases > True$. Therefore, the parser will perform "AND" operation between two variables;

- *logs: '/var/lib/mysql':* voter needs to know where access to logs. This entry assumes importance only if container logging is done into a file. In this case, a mySQL database has logging disabled by default, so parser has to check the $output > enabled$ entry. If the parameter is setted to True, it means that logging into file (or multiple files) should be handled. As a result, the parser has to integrate a configuration file that enables this feature, within the container. In this specific case of a mySQL database, it is contained in a directory, specified in the entry $storage > config$;

Finally, format for reading logs in the $outputs > logs > format$ entry should be communicated to the voter, and control over databases and tables should be enabled, with $outputs > databases$ and $outputs > tables$ entries.

Parsing operation creates a new Dockerfile, structured as follows:

```
### Starting Dockerfile ###
version: '3.9'

services:
    db1:
        image: mysql
        container_name: mysqldB1
        hostname: 'mysqldB1'
        ports:
            - "3306"
        environment:
            MYSQL_ROOT_PASSWORD: admin
        volumes:
            - 'storage1:/var/lib/mysql'
            - './conf:/etc/mysql/conf.d'

    db2:
        image: mysql
        container_name: mysqldB2
        hostname: 'mysqldB2'
        ports:
            - "3306"
        environment:
            MYSQL_ROOT_PASSWORD: admin
        volumes:
            - 'storage1:/var/lib/mysql'
            - './conf:/etc/mysql/conf.d'

    db3:
        image: mysql
        container_name: mysqldB3
        hostname: 'mysqldB3'
        ports:
            - "3306"
        environment:
            MYSQL_ROOT_PASSWORD: admin
        volumes:
            - 'storage1:/var/lib/mysql'
            - './conf:/etc/mysql/conf.d'

    tunneling:
```

```
            image: tunneling_image
            container_name: tunneling
            hostname: 'tunneling'
            restart: always
            ports:
                - "65432"
                - "3306"
            build:
                context: .
                dockerfile: Dockerfile_tunneling

        voter:
            image: voter_image
            container_name: voter
            hostname: 'voter'
            restart: always
            ports:
                - "65432"
            build:
                context: .
                dockerfile: Dockerfile_voter
            volumes:
                - 'storage1:/voter_script/log1'
                - 'storage2:/voter_script/log2'
                - 'storage3:/voter_script/log3'
        sup:
            image: supporting_image
            build:
                context: .
                dockerfile: Dockerfile_supp
            container_name: supp_container
            hostname: 'supporting'
            ports:
                - "3306"

    volumes:
        storage1:
            driver: 'local'
        storage2:
            driver: 'local'
        storage3:
            driver: 'local'
```

5.2 Correct system execution

In order to demonstrate the correct execution of the voter, the following queries have been sended to the system, in the following order:

- *$> CREATE DATABASE testdb;*
- *$> USE testdb;*
- *$> CREATE TABLE test_table (ID int, name text, primary key(ID))*;
- *$> INSERT INTO test_table (ID, name) VALUES (1, 'name1')*;
- *$> SELECT * FROM test_table*;
- *$> UPDATE test_table SET ID = 2 WHERE ID = 1*;
- *$> DROP DATABASE testdb*;

As showed from fig 5.2 to fig 5.6, all containers had work correctly, as voter logs show debugging of voting operations. When databases are created or deleted, the voter shows "*All database creation success*" or "*All database removal success*"; for table creations, after content comparisons, it shows "*All tables are equal*" and finally compares shell outputs for SELECT and USE query, showing "*All shell outputs are Equal*".

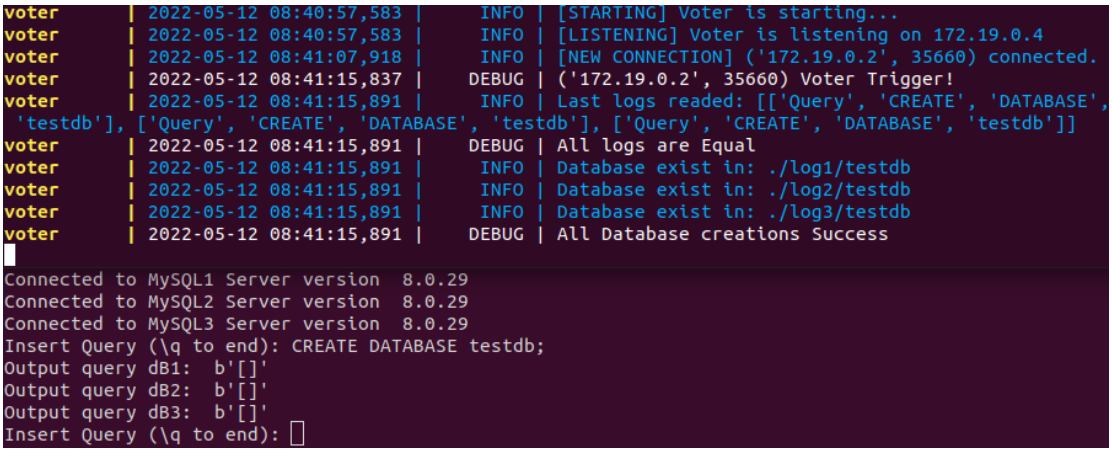

Figura 5.2. Connections to MySQL databases and CREATE DATABASE queries voting

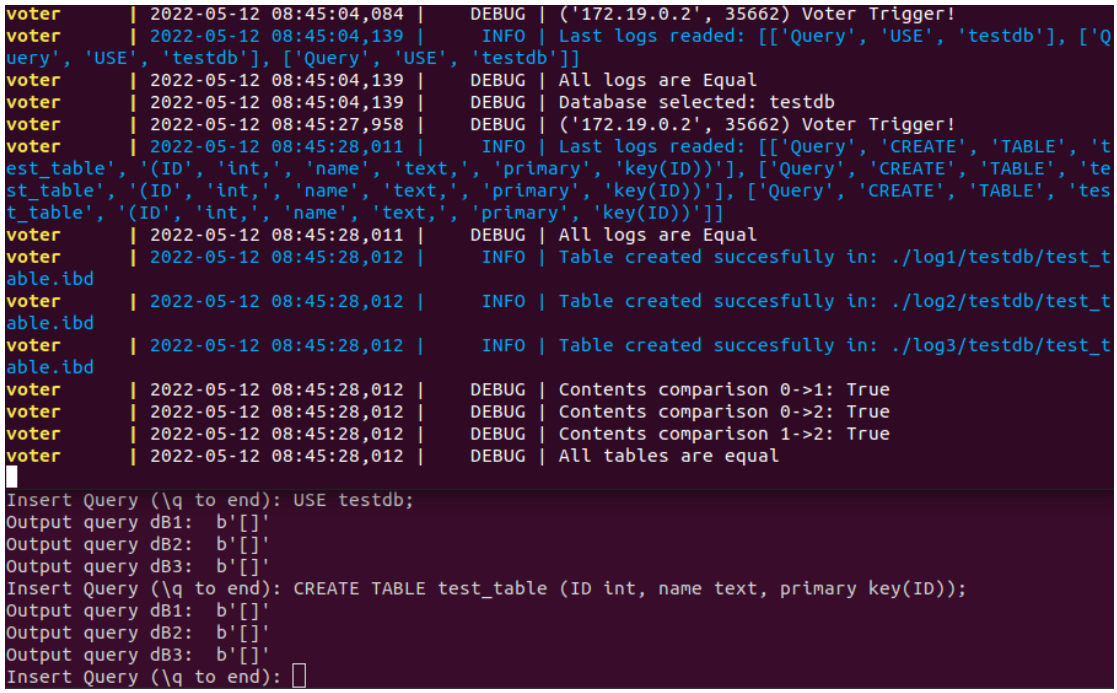

Figura 5.3. USE and CREATE TABLE queries

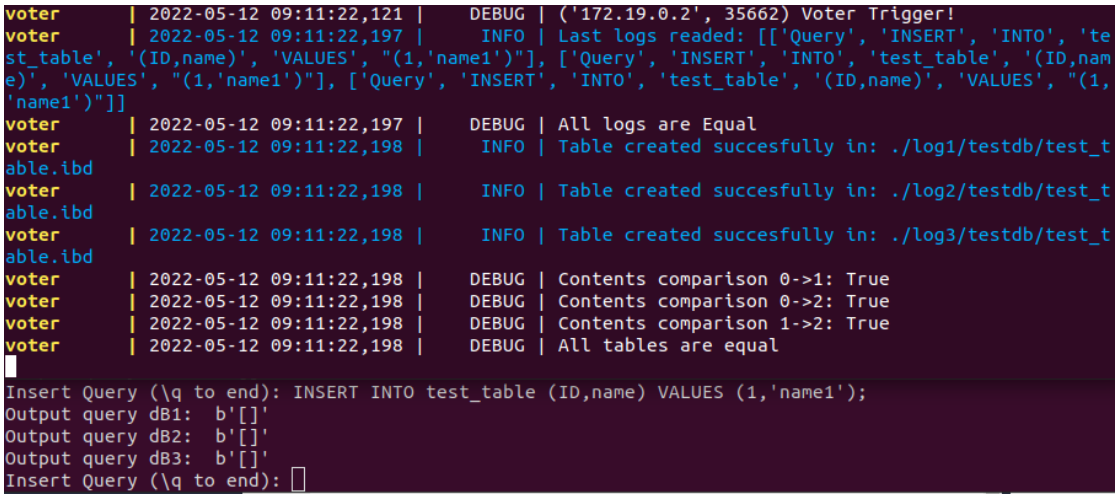

Figura 5.4. INSERT queries

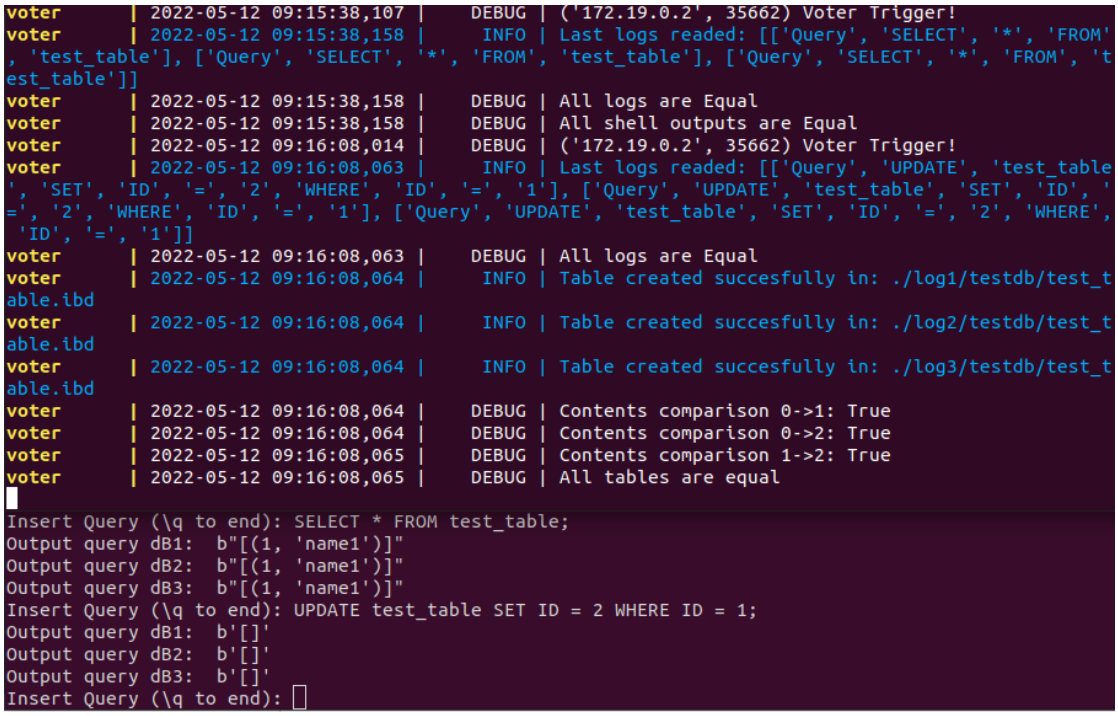

Figura 5.5. SELECT and UPDATE queries

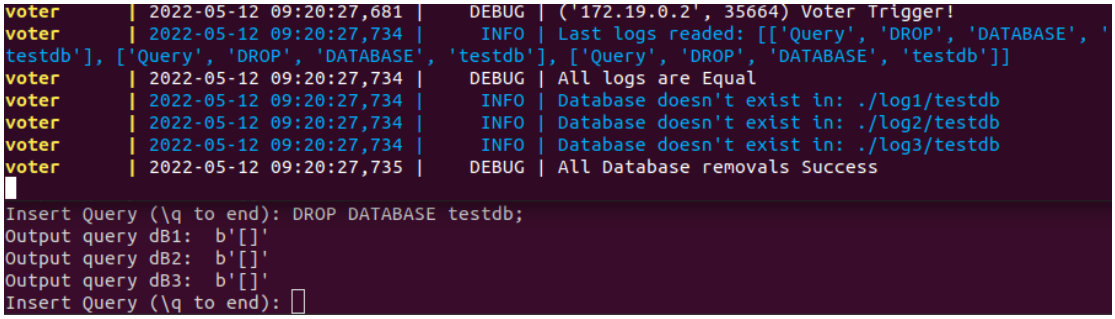

Figura 5.6. DROP queries

5.3 Fault Injections validation

The main feature of a redundancy system is to ensure a correct execution, even if failover occurs in one or more containers. In such cases, the voting operations don't get unanimity, but majority. To verify that the redundant database system works in this situation, a fault injection was performed in one database. As showed in the figs, the voter shows a warning related to

the INSERT and CREATE DATABASE queries. The voter signals, with a warning log, that the system can continue to operate, even though a fault has occurred in one of the containers. Fig 5.9, on the other hand, considers the case where all 3 containers don't create same tables in all redundant database. In this situation, the majority cannot be obtained, so the system cannot continue to run. Therefore, the voter reports this failure, requiring redeployment of the entire system.

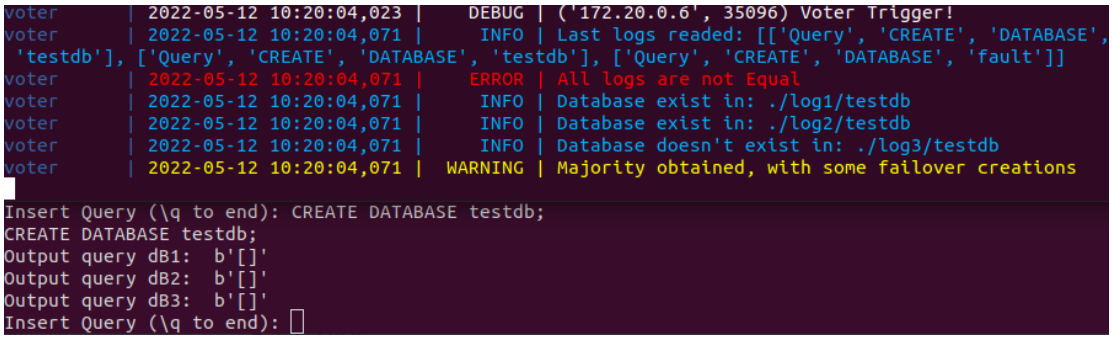

Figura 5.7. CREATE DATABASE queries with fault injection

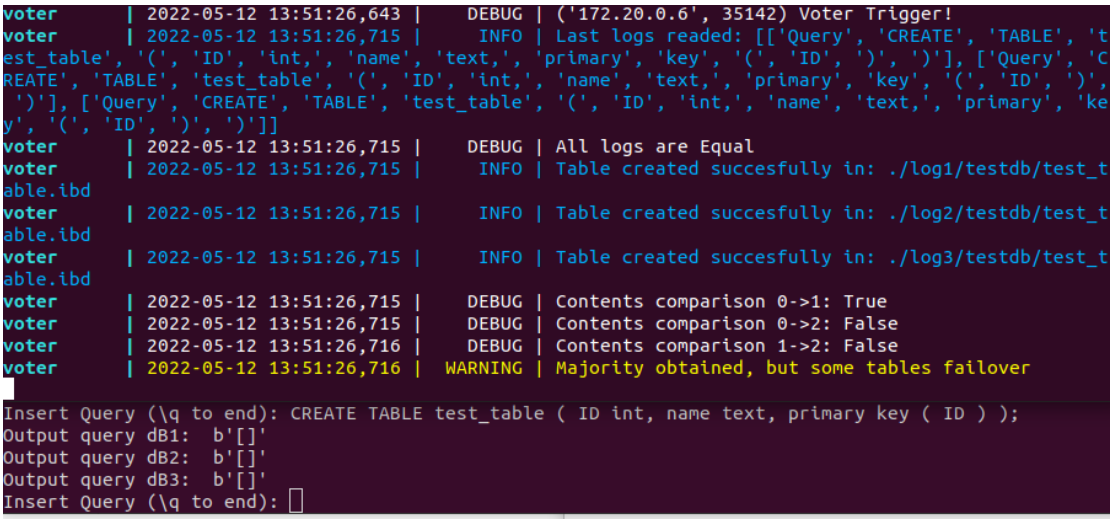

Figura 5.8. CREATE TABLE queries with fault injection

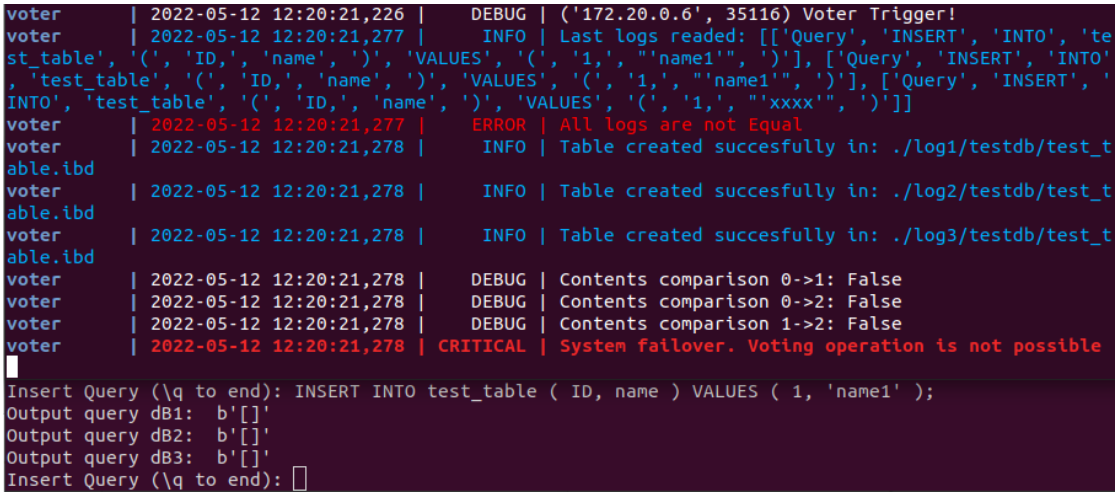

Figura 5.9. INSERT queries critical failover

Capitolo 6

Conclusions

Container classification is a useful methodology to instantiate fault tolerance systems, by acting on individual containers. A system can become more robust by introducing redundancy. Virtualization with containers is a technology that takes full advantage of such techniques, due to its flexibility and limited use of hardware resources.

However, two disadvantages emerged from this methodology:

- Facing the study of new container class, it requires a deep understanding of the software, that will run on the container. Some of these softwares, can be very complex to manage, only using features available with Docker;

- In systems where there is a large amount of deployed containers, resource consumption could be higher, if fault tolerance methodologies are adopted on the entire system. For this reason, careful resource allocation is required at the design phase;

Future improvements of this work could be studied, such as:

- Leveraging techniques for resource management, as explained in the previous chapter. Being that workload could change run-time, containers resources should increase or decrease. The adoption of these techniques should be modeled to work on the fault tolerance system;

- Elimination of the parser, by adding its functions into the Docker engine. Managers of a container system, only has to worry about compiling the fault tolerance part, and then let Docker do the deployment;

- The introduction of additional of levels of abstraction. Container classification requires a knowledge of the internal structure of containers and software. A possible approch could be a definition of default parameters, which reduces the number of input parameters required by those compiling the compose file. In this way, it's not necessary a knowledge of a container structure, simplifying the design process;

Bibliografia

[1] https://docs.docker.com/engine/

[2] https://docs.docker.com/config/containers/logging/configure

[3] J. Turnbull, *The Docker Book: Containerization is the new Virtualization*, James Turnbull, 2014.

[4] Koren, Israel, Krishna, C. Mani (2007), *Fault-Tolerant Systems*, San Francisco, CA: Morgan Kaufmann, p.3, ISBN 978-0-12-088525-1.

[5] Naruemon Wattanapongsakorn, Steven P. Levitan, *Reliability Optimization Models for Embedded Systems With Multiple Applications*, September 2004.

[6] Chunlin Yin1, Zhengyun Fang, Na Zhao, *Redundancy Mechanism of Software System and Reliability Analysis*, IOP Conf. Ser.: Earth Environ. Sci. 440 032022, 2020.

[7] M. R. Lyu, Ed., *Handbook of Software Reliability Engineering*, Mc-Graw-Hill, IEEE Computer Society Press, 1996.

[8] M. R. Lyu, Ed., *Software Reliability Engineering: A Roadmap*, IEEE Computer Society Press, June 2007.

[9] ANSI/IEEE, *Standard Glossary of Software Engineering Terminology*, STD-729-1991, ANSI/IEEE, 1991.

[10] M.R. Lyu , X. Cai, *Fault-Tolerant Software*, Encyclopedia on Computer Science and Engineering, Benjamin Wah(ed.), Wiley, 2007.

[11] Bharathi V., *N-Version programming method of Software Fault Tolerance: A Critical Review*, National conference on nonlinear systems and dynamics, 2003.

[12] A.Avizienis, *Version Approach to fault tolerant Software*, IEEE Software eg., vol- SE11, No12, Dec 1985.

[13] Kurt Kanzenbach, *Fault tolerance on the system level*, Friedrich - Alexander University Erlangen - Nuremberg.

[14] . Borkar, *Designing reliable systems from unreliable components: The challenges of transistor variability and degradation*, IEEE Micro, vol. 25,

no. 6, Nov. 2005.

[15] . Baumann, *Soft errors in advanced computer systems*, IEEE Design Test of Computers, vol. 22, no. 3, May 2005.

[16] . Narayanan and Y. Xie, *Reliability concerns in embedded system designs*, Computer, vol. 39, no. 1, pp. 118–120, Jan. 2006.

[17] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, *An experimental study of soft errors in microprocessors*, IEEE Micro, vol. 25, no. 6, Nov. 2005.

[18] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, *Fault injection for dependability validation: A methodology and some applications*, IEEE Transactions on software engineering, vol. 16, no. 2, 1990.

[19] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, *Fault injection techniques and tools*, Computer, vol. 30, no. 4, Apr. 1997.

[20] H. Ziade, R. Ayoubi, and R. Velazco, *A Survey on Fault Injection Techniques*, The International Arab Journal of Information Technology, 2004.

[21] S. K. Reinhardt and S. S. Mukherjee, *Transient fault detection via simultaneous multithreading*, In Proceedings of the 27th Annual International Symposium on Computer Architecture. ACM Press, 2000.

[22] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, *An updated performance comparison of virtual machines and linux containers*, Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On, IEEE, 2015.

[23] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht,Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt, *Improving docker registry design based on production workload analysis*, in Proc. 16th USENIX Conf. File Storage Technol., 2018.

[24] Zhuping Zou ,Yulai Xie, Kai Huang, Gongming Xu, Dan Feng, Darrell Long, *A Docker Container Anomaly Monitoring System Based on Optimized Isolation Forest*, IEEE Transactions on Cloud Computing, vol. 7, August, 2019.

[25] S. Loveland, E. M. Dow, F. LeFevre, D. Beyer, and P. F. Chan, *Leveraging virtualization to optimize high-availability system configurations*, IBM System Journal, Vol. 47, No. 4, 2008.

[26] Fawaz Paraiso, St´ephanie Challita, Yahya Al-Dhuraibi, Philippe Merle, *Model-Driven Management of Docker Containers*, IEEE 9th International Conference on Cloud Computing, 2016.

[27] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker, *Incremental Model Synchronization for Efficient Run-time Monitoring*, In Proceedings of the 2009 International Conference on Models in Software Engineering, MODELS' 09, Berlin, Heidelberg, 2010, Springer-Verlag.

[28] Russ McKendrick, *Monitoring Docker*, Livery Place, Packt Publishing, 2005.

[29] Fumio Machida, Masahiro Kawato and Yoshiharu Maeno, *Redundant Virtual Machine Placement for Fault-tolerant Consolidated Server Clusters*, Service Platforms Research Laboratories, NEC Corporation, 1753.

[30] Sarita, Sunil Sebastian, *Transform Monolith into Microservices using Docker*, International Conference on Computing, Communication, Control and Automation (ICCUBEA), Pune, Maharashtra, India, 2007.