České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská
Katedra fyzikální elektroniky

DIPLOMOVÁ PRÁCE

Bc. Jan Tomsa

Praha – 2016

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská
Katedra fyzikální elektroniky

Backup processes of data acquisition control system of the COMPASS experiment at CERN.

Diplomová práce

Autor práce:	**Bc. Jan Tomsa**
Vedoucí práce:	**Ing. Vladimír Jarý, Ph.D.**
Konzultant:	**Ing. Josef Nový**
Akademický rok:	**2015/2016**

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA JADERNÁ A FYZIKÁLNĚ INŽENÝRSKÁ

Katedra fyzikální elektroniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	**Bc. Jan Tomsa**
Obor:	**Informatická fyzika**
Školní rok:	**2015/2016**
Název práce:	**Záložní procesy řídicího systému sběru dat experimentu COMPASS v CERN** **Backup processes of data acquisition control system COMPASS experiment at CERN**
Vedoucí práce:	**Ing. Vladimír Jarý, Ph.D.**
Konzultant:	

Pokyny pro vypracování:

- Prostudujte současný stav řídicího systému sběru dat experimentu COMPASS.
- Navrhněte algoritmy pro detekci fatálních chyb, přepínání na záložní systém a synchronizaci záložního a hlavního systému.
- Navržené algoritmy implementujte a otestujte.

Literatura:

1. VIRIUS, M. *1001 tipů a triků pro C++*. Vyd. 1. Brno: Computer Press, 2011, 451, xx s.
2. KNUTH, D. E. *Umění programování*. Vyd. 1. Brno: Computer Press, 2008, xix, 648 s.
3. WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. Vyd. 1. Překlad Marek Michalek, Bogdan Kiszka. Brno: Computer Press, 2004, 351 s.
4. BODLAK, M., FROLOV V., et al. FPGA based data acquisition system for COMPASS experiment. *Journal of Physics: Conference Series*. 2014-06-11, vol. 513, issue 1, s. 012029-. DOI: 10.1088/1742-6596/513/1/012029. Dostupné z: http://stacks.iop.org/1742-6596/513/i=1/a=012029?key=crossref.78788d23de2b4a6a34d127c361123b8c
5. *The COMPASS experiment at CERN*. Nucl.Instrum.Meth. A577 (2007) 455-518.

Datum zadání:	23. říjen 2015
Datum odevzdání:	6. květen 2016

[illegible]
……………………………
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-1-

[illegible]
……………………………
děkan

V Praze 23.10.2015

Poděkování

Chtěl bych poděkovat Ing. Vladimíru Jarému, Ph.D. za vedení mé diplomové práce, podnětné připomínky a jazykovou korekturu.

Acknowledgment

I would like thank to my supervisor, Ing. Vladimír Jarý, Ph.D., for leading my diploma thesis, for his incentive notes, and language corrections.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

Declaration

I declare that I have carried out this research project myself and I have mentioned all used information sources in bibliography.

V Praze dne 6. 5. 2016 Bc. Jan Tomsa

Název práce:

Záložní processy řídícího systému sběru dat experimentu COMPASS v CERN

Autor: Bc. Jan Tomsa

Obor: Inženýrská informatika

Druh práce: Diplomová práce

Vedoucí práce: Ing. Vladimír Jarý, Ph.D.

Abstrakt

Tato diplomová práce se zabývá vytvořením záložního řídicího systému sběru dat experimentu COMPASS v laboratoři CERN. Pro běh záložního a hlavního řídicího systému je využito moderního clusterového řešení od společnosti Red Hat. Práce obsahuje detailní popis implementace a nastavení clusteru, podrobný postup přestavby řídicího procesu pro využití v clusteru a pohled na můj nově vytvořený monitorovací nástroj pro kontrolu stavu těchto procesů. V další sekci jsou dostupné testy vyvinutého řešení a v dodatku podrobný postup instalace.

Klíčová slova: CERN, DAQ, RHCL, Cluster, Python, C++, Django, Daemon

Title:

Backup processes of data acquisition control system of the COMPASS experiment at CERN.

Author: Bc. Jan Tomsa

Abstract

This diploma thesis focuses on creating backup processes of the DAQ system of the COMPASS experiment at CERN. A modern high-availability cluster solution from the Red Hat company has been used to implement the backup processes. The thesis includes a detailed description of the implementation of the cluster setup, a comprehensive guide to rebuild the control process for the needs of clustering, and an introduction to my newly developed cluster monitoring tool. In the end, there are tests of the implemented changes and an exhaustive installation instructions.

Key words: CERN, DAQ, RHCL, Cluster, Python, C++, Django, Daemon

Contents

Introduction

During the CERN shutdown during the years 2013 and 2014, both hardware and software parts of the Data Acquisition System of the COMPASS experiment has been upgraded. The new COMPASS Experiment data taking setup has been now running nearly two years. During this time a need to polish and to improve the new system has arisen.

The assignment of this diploma thesis is to increase the availability and to create a backup process of the core control process of the Data Acquisition System of the COMPASS experiment. This is achieved by deploying the control application in a high-availability cluster. The task requires some non-trivial changes to the starting sequence of the program, including a conversion to a UNIX daemon.

The first chapter shortly introduces the CERN laboratory and the COMPASS experiment in more details.

The next part of this project focuses on the introduction of the main software tools that are used to accomplish the goal of this thesis. It introduces mainly the Red Hat Cluster Suite.

The Implementation chapter includes an exhaustive description of all key parts of the newly developed software. It covers cluster setup, the transformation of the control process to a daemon, and an insight into a newly developed Cluster Monitoring Tool.

The following chapter provides the results of various tests of the newly developed software.

For those, who are interested in deploying a similar setup themselves, there is a thorough installation guide in the Appendix.

Chapter 1

Cern & COMPASS & DAQ

1.1 CERN

CERN, the European Organization for Nuclear Research, is an international physics laboratory, operating one of the largest and most complex instruments to study the basics of matter - fundamental particles. It is located north-west of Geneva (Switzerland) in the Swiss-French borderland, with the main site in Meyrin. CERN was established in 1954 by 12 European countries.

CERN provides an infrastructure and set of 6 accelerators, a decelerator, and many detectors for high-energy particle physics experiments. The largest accelerator (LHC - Large Hadron Collider) is designed to produce particle collisions with energy up to 14 TeV. Particles gain energy in the cascade of accelerators and collide at speed close to the speed of light either with fixed targets or with each other. Surrounding detectors record the results of these collisions. [1, 2]

1.2 The COMPASS experiment

COMPASS (COmmon Muon Proton Apparatus for Structure and Spectroscopy) is a high-energy particle physics experiment with fixed target situated on the M2 beamline of the Super Proton Synchrotron (SPS) particle accelerator at CERN laboratory in Geneva, Switzerland. The scientific program of the COMPASS experiment was approved in 1997. It's goal was to study the structure of gluons and quarks and the spectroscopy of hadrons using high intensity muon and hadron beams. By the year 2010 the experiment entered it's second phase COMPASS-II [3] focusing on the Drell-Yan effect, the Primakoff scattering, and the Deeply Virtual Compton Scattering. [4]

1.3 The DAQ of COMPASS

The hardware of the Data Acquisition System consists of several layers of different electronics. The layer closest to the detector is called **frontend electronics**. It's task is to capture signals directly from the detectors and convert them to digital values. There are approximately 300 000 of data channels coming from the first layer. This data is readout by roughly 250 of **CATCH**, **GeSiCA**, and **Gandalf** concentrator modules based on VME standard and grouped into subevents (i.e. partial information about the progress of particle in the detector). The readout of subevents and assembly and buffering of events is carried out by modern **FPGA cards** (Field Programmable Gate Array) which replaced original and nowadays performance-wise obsolete hardware. Data taking process is synchronized by the TCS (Trigger Control System). Full events are stored locally on hard disks and afterwards transferred to the central CERN storage facility CASTOR. [5]

The new software of DAQ is conceptually inspired by the ALICE software [6], however it is more lightweight and thus easier to use and to maintain. According to [7,8], the new software consists of five main types of processes:

A) **Master process** - a Qt console application. It is the most important part, almost all application logic is concentrated here. It serves as a mediator between Slaves and GUI. This is the main control process.

B) **Slave process** - an application that controls and monitors a custom hardware (e.g FPGA, ...). It is controlled and configured by the Master, it informs the Master process about the state of the hardware it is deployed on.

C) **GUI** - a Qt GUI application designed for controlling and monitoring of the whole DAQ. It sends commands to the Master. Master sends back monitoring data about hardware controlled by Slaves. GUI can run in many instances, only one has the rights to change configuration and to execute control commands, the others are only allowed to monitor the status of DAQ.

D) **Message Logger** - a console application that receives informative and error messages and stores them into the MySQL database. It is directly connected to the Master and to the Slave processes via the DIALOG Communication Library services

E) **Message Browser** - a GUI application that provides an intuitive access to messages from system (stored in the database) with an addition of on-line mode (displaying new messages in realtime). Equipped with filtering and sorting capabilities, it is able to run independently from the whole system in case of emergency.

Chapter 2

Software technologies

2.1 Red Hat Cluster Suite

This section will be focused on detailed description of **Red Hat Cluster Suite** (RHCS) used to deploy the **master process** in the **highly-available** mode.

High-availability cluster (also known as HA cluster or fail-over cluster) is a formation of two or more computers working together to perform a task or to provide an uninterrupted (with minimal down-time) run of core services or server applications. It is operated by so called High-availability software, which manages server groups (clusters) and utilises redundant resources when any system components fail. Should such failure of a server running a particular application occur without cluster setup, the application will be unavailable until the problem is identified and fixed and server restarted. The high-availability cluster overcomes this issue by detecting both software and hardware faults and instantly relocating afflicted services and applications to redundant, healthy nodes [1] without any required interference from system administrator. This process is called **failover**. The target node may be automatically configured prior to starting the service (e.g. mounting filesystems, starting additional supportive services and applications, . . .). The HA cluster is also trying to eliminate single points of failure. [9, 11]

Red Hat Cluster Suite is a set of software tools that can be set up in a variety of configurations including high performance, high availability, load

[1]In the context of this thesis, a node is a single computer (server) within the cluster formation.

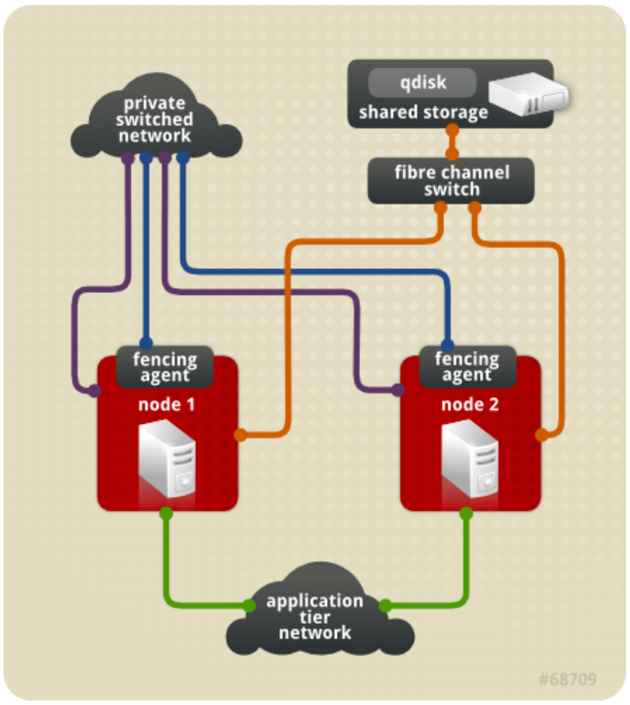


Figure 2.1: An ideal two-node cluster. [10]

balancing, etc. Following main parts comprise the RHCS: [11]

- Cluster infrastructure - fundamental tools for nodes to build a cluster; configuration files management; cluster membership management; fencing
- High-availability Service Management - failover of services
- Cluster administration tools - configuration of management tools used to set up, configure, and manage Red Hat cluster.

2.1.1 Concepts and Components

We could dive right into cluster configuration, it isn't that difficult, but without fully understanding how all the parts work together, it would be near to impossible to set up this fairly complex system correctly.

Quorum

In the context of clusters, quorum is a synonym for majority. In simplicity, it is defined as the minimum number of nodes required to provide clustered services. However all nodes are assigned a number of votes and the cluster is told how many votes to expect in total. The default algorithm of RHCS for determining quorum is called **Simple Majority Quorum** which demands that more than half of the expected votes must be put together by the online nodes to gain quorum and form a cluster.

Sometimes a device with assigned number of votes (so called **quorum disk**) can be added to the cluster. In that case, the expected votes is the sum of all node's votes plus the votes of the quorum disk. During a split, each part adds up votes of itself plus the votes of the quorum devices it can communicate with.

In the case the nodes spit into two or more parts (for example due to network failure), the quorum comes to play. Which ever part has the quorum (majority) can form a new cluster and safely start providing clustered services, because the other parts know that they don't have quorum and won't do anything. The winner part then fences the lost nodes.

The requirement for the majority of votes is crucial. If the exact half of votes sufficed and the cluster split into equally sized groups, it would lead to a split-brain situation. For example, a cluster of four nodes could split into two groups by two. In this case, both groups would gain quorum (2 votes) and might try to take over the cluster and start providing services. It would lead into disastrous scenario.

However, as always, there is an exception to the rule. In the case of two node cluster, any failure results into equal split 1+1 and no node gains quorum, thus there is no high availability. Considering this, we don't want to enforce quorum. The downside is that the consistency of the cluster is now only assured by fencing.

Nevertheless, a proper quorum can be achieved even in the two node cluster by using a quorum disk. But we will not be using the quorum disk in this project, it is not necessary and it would increase the overall complexity.

CMAN

CMAN is an abbreviation for **C**luster **MAN**ager. It is a distributed cluster manager and runs on each node (as a service) and acts as a quorum provider. It sums up the member votes of the cluster and decides if it has the majority. If if does, the cluster is "quorate" and is allowed to provide cluster services. CMAN also stars and stops all required services needed for cluster operation.

Corosync and Totem

Corosync is the core of the cluster. It mediates communication between nodes. It takes care of cluster membership, message passing and quorum. It uses a totem protocol for "heartbeat" monitoring of the other nodes. It passes token around the nodes. Once a node receives a token, it can send and receive messages, and when it finishes, it passes the token to the next node. If the token is not passed in time ($\sim 238ms$ by default), an error counter is increased and a new token is generated. If too many tokens are lost in a row, a node is marked as dead and the rest of the cluster is informed about the new topology. The totem protocol supports also so called "rpp" (Redundant Ring Protocol). It can be used to add a backup ring for passing token on a separate network in case of the failure of the primary ring.

Fencing

Fencing is a critical part of clustering. It is a way of bringing (lost/dead) node to a known state, to a state where it can not affect cluster resources and provide services. When a node stops responding, the node is declared dead approximately after one second (the default timeout is roughly 238ms, the error count limit is 4 by default). The cluster checks if it still has the quorum. If it does, the cluster software freezes, the dead node withdraws from the cluster and the corosync calls **fenced** to fence the dead node.

A fence device is a device that is able to fence a particular node. Fencing can be accomplished in a variety of ways:

- Power fencing - disconnecting power via remotely controlled power switch
- Fabric fencing - disconnecting network and storage via network switch
- HP iLO and similar management interfaces

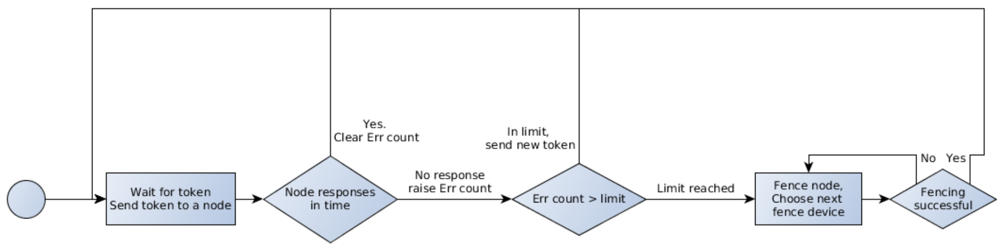


Figure 2.2: A fencing process diagram.

- Fencing virtualized machines - via virtualization host; KVM, virtualbox, VMware . . .

A fence agent is a script that can controll related fence device. The fenced daemon fetches information about all fence devices assigned to the dead node. It then iterates through them and calls appropriate fence agents with parameters valid for the dead node. It does so until one of the fence agents reports success. During this time **the cluster is effectively hung.** The figure 2.2 summarizes the fencing process.

The normal cluster operation is restored. The rgmanager relocates available services from the dead node to the rest of the nodes that formed a new cluster.

In the scenario of two node cluster, the cluster may split into two separated nodes. Both can be still alive but can't see each other. In that case they will start fencing each other, the faster one winning and forming a new cluster.

It is important to note that the fence devices must be properly configured, otherwise the cluster might never fence the dead node and thus remain in a blocked state.

RGManager

The RGManager (Resource Group Manager) is a service (daemon) that takes care of cluster resources and services. It controls their starting, stopping, migrating, relocating, and recovering. The RGManager is based on three concepts: Failover Domains, Resources and Services. [12, 13]:

- Failover Domain is a set of nodes which a service can run at. It allows to configure a set of rules defining service migration policy, preferred nodes for services, etc.

- Resources are building blocks of services. They can be used to create resource trees. Resources are of various types, for example:
  - File system
  - IP address
  - service, script
  - virtual machine
- Services group resources together. They can be hierarchically dependent on each other, or scheduled to start simultaneously. They can be assigned to failover domains. Services are what is being started, stopped, migrated, ...

The rgmanager daemon is started separately from the cman. It implies that to start a cluster to the fully operational state both cman and rgmanager daemons (in that order) need to be started.

The main configuration for the cluster is the `cluster.conf` file located at `/etc/cluster/`. It uses `XML` language to configure the topology and properties of the cluster. It can contain tags for configuration of all cluster components (cman, rgmanager, fencing, ...).

2.2 C++ and Qt framework

C++ is a general-purpose programming language. It is compiled and strongly typed, it offers object oriented, imperative, and multi-paradigm programming features. It's popularity is supported by huge library support and backward compatibility with C. It first appeared in 1983, written by Bjarne Stroustrup. The current C++ standard is C++14. [14,15]

The Qt is a C++ framework for developing applications for a wide spectrum of platforms including Windows, linux, OS X, Android, etc. The first version has been released in 1995 under the patronage of a Norwegian company Trolltech. It offers a straightforward way of creating GUI (Graphical User Interface) applications with the help of QtCreator, QtDesigner, and other useful tools. Also command line applications can be developed with the same ease. Qt enriched the C++ language with many classes for creating graphical user interface (QWidget), for using network, databases, ... The key feature it also added was a **signals & slots** mechanism for an independent inter-object communication between Qt objects. Although the Qt is written in C++, bindings for many languages are available (Python, Java, ...) [16]

2.3 Python

Python is a high-level, object-oriented, general-purpose, interpreted, dynamic programming language. It supports many programming paradigms - imperative, function, procedural, object-oriented. It is strongly and dynamically typed. It utilises a garbage collector. It is highly readable, using whitespace indentation to delimit blocks. It's usage scenario ranges from simple scripts, through scientific programs to backends of websites.

Python has been designed by Guido van Rossum and first released in 1991. Currently, there are two stable releases - versions 2.7.11 and 3.5.1 (December 2015). [17]

2.4 Django

Django is a python framework for web development. It is safe to say that django uses MVC (Model-View-Controlled) design pattern. Django includes ORM (objec-relational mapper) for database-driven web applications, a standalone development web server, a support for internationalization and much more. Django was first released in July 2005. [18]

2.5 DIALOG Communication Library

As the name suggest, the `DIALOG Communication Library` is a library that mediates communication between all parts of the COMPASS DAQ setup. It has been developed by Ing. Ondřej Šubrt during the spring of the year 2016. It replaced previously used `DIM` library (Distributed Information Management, [19]). It is written in Qt, allowing easier integration into our DAQ system. It is conceptually based around the same paradigms as the DIM. It uses the similar concept of a publisher-subscriber communication with the addition of a Control server which stores information about all published services and available commands (Figure 2.3).

Similarly to DIM's DID (DIM Information Display), the DCL provides an inspection tool called `DIALOG Communication GUI` which can be used to browse all registered processed and look through the services they provide and they are subscribed to.

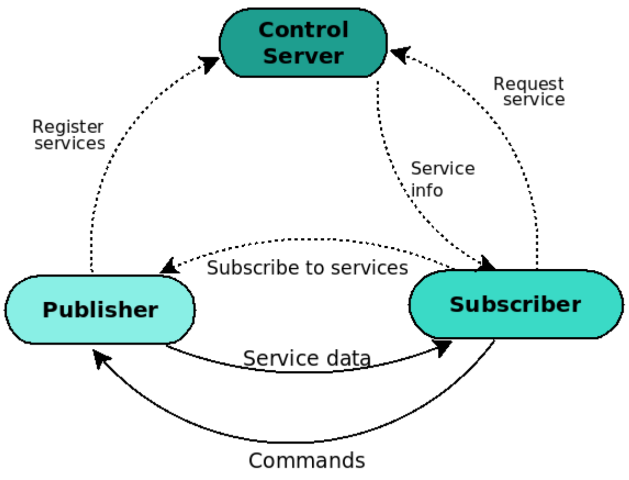


Figure 2.3: The DIALOG Communication Library concept.

Chapter 3

Implementation

3.1 Cluster configuration

The main cluster configuration [20] file is located at `/etc/cluster/cluster.conf`. It is an XML file.

3.1.1 Cluster, cman, nodes, fencing, totem

Everyting is enclosed in a `<cluster>` tag. The tag has a `name` property which defines the cluster name, it must be unique in our network. The second compulsory parameter we use is a `config_version` parametr which stores the current version of the configuration file. It is required to increment the number of version upon every change, otherwise the cluster would not know the file needs to be reloaded. We can start from the number one.

CMAN

Next we must configure the cman. Because we are creating a special kind of cluster - a two node cluster - we must use an appropriate `two_node` parameter of the `<cman>` tag. Usually the `expected_votes` parameter is set automatically (the quorum requires 50% + 1 votes), but in this special case scenario it must be set manually to 1. This effectively disables quorum.

Nodes

Cluster nodes are defined by `<clusternode>` tags inside a `<clusternodes>` wrapper. Each node has it's `name` attribute. It specifies the name. It is advisable to set the node name to match the fully qualified domain name (FQDN) which is resolved to the IP address the node will have. The bond between the FQDN and the IP address can be for example set in `/etc/hosts`. The node names should be chosen such that they fit into the naming convention of the place the cluster is deployed at. The second attribute, `nodeid`, is a number and it should be unique within the cluster. The current form of the configuration file is as follows:

```
<?xml version="1.0"?>
<cluster name="master_cl" config_version="1">
  <cman expected_votes="1" two_node="1"/>
  <clusternodes>
    <clusternode name="vm1.dp.jt" nodeid="1"/>
    <clusternode name="vm1.dp.jt" nodeid="1"/>
  </clusternodes>
</cluster>
```

Listing 3.1: cluster.conf with nodes defined.

Fencing

Fencing is a process of removing a node from cluster, putting it into a known (turned off) state, using fence devices. There are many fencing devices, as mentioned earlier in the section 2.1.1. As this project has been developed and is intended to be deployed using virtual machines, the only fence device we need has to be able to connect to the virtual host and kill the particular virtual machine. Fence agents are scripts located in `/usr/sbin/`. They serve as a mediator between the fenced daemon and the fence hardware. For virtual fencing there are scripts already prepared for virsh (`fence_virsh`) and VMware (`fence_vmware`). The script for VirtualBox (`fence_vbox`) should be also a part of the fence-agents package. It's copy is on the enclosed DVD, just for sure. Other custom fence agents can be created using provided python `fencing` module that uses the FenceAgentAPI (for more see [21])

Fence devices are defined using `<fencedevice>` tags wrapped by `<fendedevices>` parent. Each fence device expects following attributes to be defined:

- `name`: a custom name

- `agent`: a name of the script that should be used, located in `/usr/sbin/`

The `fence_vbox` agent additionally requires login credentials to the virtual host machine

- `ipaddr`: a hostname or an equivalent IP address of the physical device that the script will connect to
- `login`: a name of user allowed to control VirtualBox (root or an user in the vboxusers group)
- `passwd` or `identity_file` or `passwd_cript`: depending on preferred SSH authentication method, one of these options should be used and appropriate values inserted (password, path to identity file, path to a script to retrieve password)

The usage of previously defined fence devices is declared within each cluster node. In the `<clusternode>` tags a `<fence>` section is created. It contains a list of fence methods to use when fencing a node. Each fence method consists of one or more `<device>` tags that link (via the `name` attribute) `<fencedevice>` with additional attributes. These attributes tell the fence daemon how to use the particular fence device to kill an appropriate node. The `port` corresponds with the virtual machine name defined withing the VirtualBox, `action` tells the fence device whether to power off or restart the machine when being fenced.

The `delay` parameter should be noticed. It is a **critical** option within the two node cluster setup. It specifies, how many seconds to wait before fencing the node. In the case of network failure, both nodes will be alive and if they don't see each other, they will try to fence each other. With the delay set, it is assured, that one node will have a head-start in fencing. Without the delay, both could be fenced simultaneously, which obviously would be an undesired result.

The `<fence_daemon>` tag and it's attribute `post_join_delay` configures the fence daemon to wait specified amount of seconds for other nodes when the cluster is starting. It can happen that the nodes are not powered on at the same time, or that for one the booting takes longer than for the rest. Without this delay, the slower nodes could be fenced before they are even able to boot up and start cluster daemons. For development purposes it is better to set the delay to larger value (like 30s).

Totem

The `<totem>` tag configures the Totem protocol. Setting `rpp_mode` to "none" turns off the redundant ring protocol and setting the `secauth` to "off" turns off encription of the cluster communication, which is not required in closed private network and it is simpler to setup and faster.

The code snipped bellow (3.2) shows the current status of the `cluster.conf` file. (Notice the `config_version` number incremented)

```
<?xml version="1.0"?>
<cluster name="master_cl" config_version="2">
 <cman expected_votes="1" two_node="1"/>
 <clusternodes>
  <clusternode name="vm1.dp.jt" nodeid="1">
   <fence>
    <method name="vbox">
     <device name="VBoxMan" port="vm1_dp" action="reboot"/>
    </method>
   </fence>
  </clusternode>
  <clusternode name="vm2.dp.jt" nodeid="2">
   <fence>
    <method name="vbox">
     <device name="VBoxMan" port="vm2_dp" action="reboot" delay="10"/>
   </method>
   </fence>
  </clusternode>
 </clusternodes>
 <fencedevices>
  <fencedevice name="VBoxMan" agent="fence_vbox" login="user"
                 ipaddr="vboxhost.dp.jt" passwd="pass"/>
 </fencedevices>
 <fence_daemon post_join_delay="6"/>
 <totem rrp_mode="none" secauth="off"/>
</cluster>
```

Listing 3.2: cluster.conf with fencing defined.

3.1.2 Resources, fail-over domains, services

There is a `<rm>` tag (a shortcut for resource-manager) in the `cluster.conf` which contains resources, fail-over domains and services.

Resources

All available resources are defined within the `<resources>` element. Resources are used in Services and referenced back to their definition here. Within this cluster, two resource types are used:

- `<ip>`: This represents an IP address. When a service uses this resource, the node which the service runs at is accessible via the specified IP address.
- `<script>`: This is a shortcut to a executable file representing (usually) a daemon. The `file` attribute sets the path to the script; the `name` attribute is there for referencing this resource

We use one IP resource, this will always be used together with the main master process so it is accessible on the same address whichever node it is currently deployed on. Two custom scripts, `master_control.py` and `master_backup.py` are used. The first runs master in the normal mode, the second runs master in the backup mode.

Fail-over Domains

Fail-over domains control which nodes and under which circumstances services may run at. Each fail-over domain contains a list of nodes they are a member of. Fail-over domains have following parameters:

- `name`: an unique name of the domain, used by services to choose which domain they are a part of
- `nofailback`: this tells the cluster not to fail back any services. If turned on, services won't migrate back to their previous node should the node return to the cluster.
- `ordered`: allows to set preferences between nodes within the domain.
- `restricted`: if turned on, it allows to run services only within the domain. With no nodes alive in this domain, services cannot be started. If the domain is unrestricted, services are allowed to run outside of the domain if no domain members are alive.

In this project, three fail-over domains are used. One is for the master process and for the IP address. This domain called `master_ip_domain`, operates on both nodes, nofailback is turned on, and it is ordered and unrestricted. Services in this domain therefore start on the node with the higher priority, they are allowed to migrate to other nodes when the main node dies, but the services will not migrate back to the first node when it rejoins the cluster.

There are additional two domains, `backup_domain_1` and `2`. They are ordered, restricted, and with no failback. Each of them is only on one node. They are for the master backup.

Services

The `<service>` element contains references to the resources. They can be both parallel and serialized. When in parallel, they start simultaneously, when in series they are dependent, the nested are started only after the parents are running. Services are assigned to domain. The `exclusive` tag set to zero (`0`) specifies that a given node is allowed to run also other services next to this one. The `recovery` attribute tells what to do when service fails.

The `master_ip` service deserves attention. It is defined as a resource tree, IP being parent, and a script controlling the master as child. This configuration ensures that the master process will be always accessible via the same IP address. This makes the master process look like it still runs on one computer although it can migrate between the two cluster nodes.

Rgmanager regularly checks the status of resources, the default value being 30s. If we want to change this interval, a value in the script resource executable located in `/usr/share/cluster/script.sh` needs to be adjusted. The minimal value is 10s.

The resource part of the `cluster.conf` can be seen in listing 3.3.

```
<?xml version="1.0"?>
<cluster name="master_cl" config_version="2">
...
 <rm>
  <resources>
     <ip address="10.0.0.10" monitor_link="on" sleeptime="10"/>
     <script file="/path/to/master_backup.py" name="master_backup"/>
     <script file="/path/to/master_control.py" name="master_control"/>
  </resources>
  <failoverdomains>
     <failoverdomain name="master_ip_domain" nofailback="1"
```

```
                    ordered="1" restricted="0">
        <failoverdomainnode name="vm1.dp.jt" priority="1"/>
        <failoverdomainnode name="vm2.dp.jt" priority="2"/>
      </failoverdomain>
      <failoverdomain name="backup_domain_1" nofailback="1"
                    ordered="1" restricted="1">
        <failoverdomainnode name="vm1.dp.jt" priority="1"/>
      </failoverdomain>
      <failoverdomain name="backup_domain_2" nofailback="1"
                    ordered="1" restricted="1">
        <failoverdomainnode name="vm2.dp.jt" priority="1"/>
      </failoverdomain>
    </failoverdomains>
    <service name="master_ip" autostart="1" domain="master_ip_domain"
             exclusive="0" priority="1" recovery="restart">
      <ip ref="10.0.0.10">
        <script ref="master_control"/>
      </ip>
    </service>
    <service name="master_backup_1" autostart="1" domain="backup_domain_1"
             exclusive="0" priority="2" recovery="restart">
      <script ref="master_backup"/>
    </service>
    <service name="master_backup_2" autostart="1" domain="backup_domain_2"
             exclusive="0" priority="2" recovery="restart">
      <script ref="master_backup"/>
    </service>
  </rm>
</cluster>
```

Listing 3.3: cluster.conf resources.

3.2 The Master Process

The master process has always been running as a regular console application. As the result of this project, the master needs to transformed. Master must be able to

- run in background - daemon
- be controlled via start, stop, restart, status commands
- determine it's status when running in background
- run as a master

- run as a backup
- synchronize master to backup
- run both standalone and in cluster

3.2.1 Transformation to a daemon [1]

A daemon is a process that runs in the background, without interacting with user and that doesn't belong to any terminal session. [22, 23] Daemons are usually controlled via scripts that can start, stop, or check the status of the daemon.

A function called `daemonize` with following definition (the input parameters specifies the name of the daemon, a working directory and files where to redirect standard input, output, and error output) has been added to the master:

```
int daemonize(QString name, QString path, QString outfile,
                    QString errfile, QString infile);
```

The daemonization process starts with forking. The `fork()` function, when executed, spawns a child process. The child process is the exact copy of the parent process. In code, the child and the parent process can be distinguished by the return code of the `fork()`. The parent process receives the PID (Process ID) of the new child process, whereas the child process receives zero. If the fork failed, a negative number is returned. When successful, the parent process is allowed to end. This returns control to the shell invoking the program. This guarantees that the child process will not become a process group leader. `setsit()` would fail if the process was a process group leader.

The next step is to call `setsid()`. The child process becomes a process group and session group leader. A controlling terminal is associated with a session and this new session doesn't have any terminals assigned, which is a desired state.

Then a second `fork()` is called. The parent process that is a session leader exits and the new child, a non-session group leader, can never regain a controlling terminal.

[1] Please note that this is not a tutorial for joining the dark side. A daemon should be distinguished from a demon, which is an evil spirit or a devil in some religions.

```
pid_t child_pid;
//FIRST FORK
child_pid = fork();
if (child_pid < 0){ //fork failed
    std::cerr << "Failed to fork" << std::endl;
    exit(EXIT_FAILURE);
}
if (child_pid > 0) { //only parent
    exit(EXIT_SUCCESS);
}
if (setsid() < 0) { //failed to become session leader
    std::cerr << "Failed to become session leader" << std::endl;
}
//SECOND FORK
child_pid = fork();
if (child_pid < 0){ //fork failed
    std::cerr << "Failed to fork" << std::endl;
    exit(EXIT_FAILURE);
}
if (child_pid > 0) { //only parent
    exit(EXIT_SUCCESS);
}
```

Listing 3.4: A code snippet of the double-fork (magic).

The next step is to call `chdir(path)`, where the `path` is the working directory we want the daemon to run in. It is a good practise to set it to "/" so that it doesn't keep any directory (which for example an administrator would like to unmount) in use.

`umask(0)` is called in order to have all permission for anything the program writes.

The next thing to do is to close the standard input, output, and error output file descriptors. Optionally, we can then open new outputs and redirect them to some logging files, for example.

```
//CLOSING ALL FILE DESCRIPTORS
fclose(stdin);
fclose(stdout);
fclose(stderr);

//REOPEN stdin, stdout, stderr
*stdin = *fopen(infile.toStdString().data(), "r");
*stdout = *fopen(outfile.toStdString().data(), "w+");
*stderr = *fopen(errfile.toStdString().data(), "w+");

//OPEN SYSLOG
```

```
openlog(name.toStdString().data(),LOG_PID,LOG_DAEMON);
```

Listing 3.5: A code snippet of closing and reopening filedescriptors.

3.2.2 Controlling via start, stop, restart

A proper daemon can be controlled via a script using additional parameters like start or stop. To implement this functionality, we must be able to send signals to the running background process. For such a task, the daemon's PID needs to be known. Therefore the daemon must write it's PID into a file once it is in the daemonized state. [listing 3.6]

```
int writePidFile(){
    QFile file(pidFile);
    if (file.open(QIODevice::ReadWrite)) {
        QTextStream stream(&file);
        stream << getpid() << endl;
    } else {
        std::cerr << "PID file " << pidFile.toStdString().data()
        << " can't be opened. Exiting." << std::endl;
        file.close();
        return 1;
    }
    file.close();
    return 0;
}
```

Listing 3.6: Writing a PID

The controlling part of the master process checks for input command line arguments (start, stop, status, etc) and calls appropriate functions to handle the desired task. We can take a look at the `daemon_start()` function in listing 3.7.

```
void daemon_start(){
  __pid_t pid = readPidFile();
  if (pid > 0){ //pidfile exists and contains valid PID
    if (isPidAlive(pid)){ //if process is with pid is running
      std::cout << "Daemon currently running with pid " << pid << std::endl;
      exit(EXIT_SUCCESS); //
    } else {
      std::cout << "Daemon not running, but pidfile present.
                 Non-graceful shutdown." << std::endl;
      delPidFile();
    }
  }
  int res = daemonize("master_daemon", "/", "/dev/null", "/dev/null", "/dev/null");
  if (res == 0){
    daemon_run();
  } else {
    std::cerr << "Daemonization failed." << std::endl;
    exit(EXIT_FAILURE);
  }
}
```

Listing 3.7: A function starting the daemon.

This function first checks if the daemon is running by checking the existence of PID. If the process is running, it does nothing and returns success. If the daemon is not running, it starts the daemonization process and then starts the event loop of the master process. The `readPidFile()` function is similar to the `writePidFile()`, but reads from the pidfile, returning greater-than-zero number (PID) if the pidfile exists and contains valid PID number.

The existence of a valid pid number doesn't mean that the process is still running. The process could exit abruptly without cleaning the pidfile. For this case, there is a function called `isPidAlive(__pit_t pid)` [3.8] that tries to determine if a process with given pid number exists. It uses a system function `kill(pid, signal)` that can send signals to a process with given pid. Signal 0 does nothing, but error checking is still performed. This allows for determining whether the process is alive or not. If the return code of the `kill` is zero, the signal was delivered and the process is alive. Otherwise the signal could not be delivered and a variable `errno` is set depending on the error. If the error is ESRCH, the process doesn't exist. If the error is EPERM, the process is alive, but our program does not have permission to send the signal to this process. If the error is something else, the most-used technique is to assume that the process does not exist.

```
bool isPidAlive(__pid_t pid){
    if (pid > 0){
        //signal 0; does nothing
        //but allows to check if process with pid is alive
        int res = kill(pid, 0);
        if (res == 0){
            return true;
        } else {
            if (errno == ESRCH){
                //no process with pid pid
                return false;
            } else if (errno == EPERM) {
                //process alive, but we don't have permission
                return true;
            } else {
                //we have a problem :D
                return false;
            }
        }
    }
}
```

Listing 3.8: A function determining if a process with PID pid exists.

The function `daemon_stop` (listing 3.9) is used to stop the running daemon. Similarly to the `daemon_start` it first checks whether the daemon is running. If not, it's job is done. If it is running, it tries to gracefully shutdown the daemon. The daemon catches SIGTERM signals to do a clean up (free memory up, delete pidfile, etc). There is a timeout of 10s in which the daemon is periodically sent a SIGTERM until it exists or timeout passes. If the timeout passes and the daemon is still running, an uncatchable sginal SIGKILL is send. SIGKILL should kill the process immediately (if we have permission to do so). The figure 3.1 summarizes the procedure.

```
void daemon_stop(){
    __pid_t pid = readPidFile();
    if (pid > 0){ //pidfile exists and contains valid PID
        if (!isPidAlive(pid)){ //if process is with pid is running
            //success, process not running, no need to stop it
            delPidFile();
            return;
        } else {
            //killing
            float timeout = 10; //keep trying to kill only for 10s
            int ret = 0;
            while (timeout > 0){
                timeout -= 0.1;
                //SIGTERM is catched and allows for graceful shutdown.
                ret = kill(pid, timeout > 0 ? SIGTERM : SIGKILL);
                if (ret == -1 && errno == ESRCH){
                    //process is dead
                    delPidFile();
                    return;
                }
                usleep(100000);
                if (timeout <= 0){
                    usleep(200000); //additional time for the last kill
                    if (isPidAlive(pid)) {
                        exit(EXIT_FAILURE);
                    } else {
                        delPidFile();
                        return;
                    }
                }
            }
        }
    } else {
        std::cout << ”No pid file. Is daemon already stopped?” << std::endl;
        return;
    }
}
```

Listing 3.9: daemon_stop function.

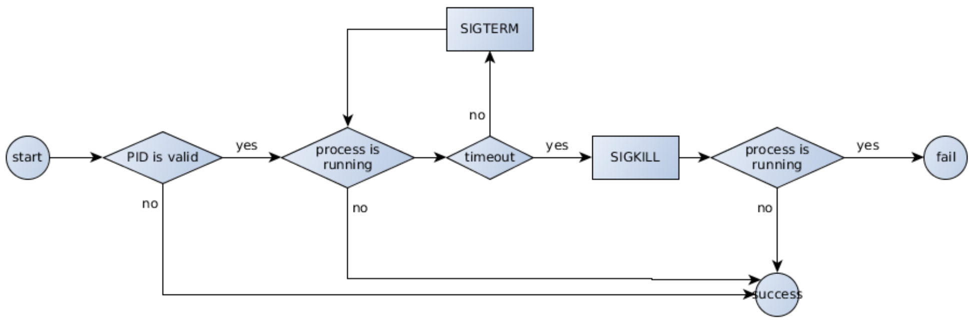


Figure 3.1: A diagram of the daemon_stop function.

A function to detect the daemon status, `daemon_status` [3.10] is very simple, it detects whether the daemon is running. It exists with correct exist codes, depending on the state of the daemon.

```
void daemon_status(){
  __pid_t pid = readPidFile();
  if (pid > 0){ //pidfile exists and contains valid PID
    if (isPidAlive(pid)){ //if process with pid is running
      std::cout << ”Daemon currently running with pid ” << pid << std::endl;
      exit(EXIT_SUCCESS); //
    } else {
      std::cout << ”Daemon not running, but pidfile present.” << std::endl;
      exit(1);
    }
  } else {
    std::cout << ”Daemon not running. No pid file” << std::endl;
    exit(3);
  }
}
```

Listing 3.10: daemon_status function.

All return codes from the master process (and from the daemon control part) are LSB compliant [24], as the rgmanager of the cluster requires.

3.2.3 Master and backup, synchronization

One of the requirements on the master process is to able to run in two modes. One mode is the active and fully working master, the second mode is a waiting backup mode which is always ready to switch to the full version. When the master is started, it automatically starts to the backup mode. In this mode it waits for the `SIGUSR1` signal, which switches the daemon to the master mode. The signal is emitted when the control part of the master is called with parameter `master`. When the signal is catched, a handler `maste_mode` is called. It finishes threads in the backup mode and starts the master mode.

```
void master_mode(qint32 sig){ // switch to master mode
    if(!masterMode) {
        syslog(LOG_NOTICE,"Switching to master mode. SIGUSR1 received.");
        finish_backup();
        masterMode = true;
        init_master();
    } else {
        syslog(LOG_NOTICE,"SIGUSR1 received. Already in master mode.");
    }
}
```

Listing 3.11: A function that switches the daemon to master mode.

The synchronization of the master and the backup master is resolved by using the DIALOG Communication Library. The full master mode already uses this library for coordinating the DAQ system. To send synchronization information, a new service `MASTER_SYNC` has to be registered. It is done by calling

```
server->registerServiceSlot("MASTER_SYNC");
```

inside the `run()` function of a `SenderProcessorThread` instance. Then we need to setup a timer which regularly triggers a function that updates the synchronization data and pushes that through the service. This (listing 3.12) is just an example, a presentation of a framework allowing to send arbitrary data to the backup process.

```
void SenderProcessorThread::MSyncServiceSlot(){
  QString curr_time = QDateTime::currentDateTime().toString("hh:mm:ss");
  QByteArray mess;
  mess.append(QString("Syncing master at: "));
  mess.append(curr_time);
  emit sendServiceMessageSignal("MASTER_SYNC", mess);
}
```

Listing 3.12: Sending synchronization data.

The backup process needs to subscribe the `MASTER_SYNC` service to receive the synchronization data. The function `init_backup` initializes all components required to connect to the DIALOG Control Server. A new class `MSyncReceiverProcessorThread` has been created. It handles subscribing of the service and receiving of messages.

```
server->requestServiceSlot("MASTER_SYNC");
```

When new message arrives, a message handler `messageReceivedSlot` is called which (in this synchronization example) prints the received message to the redirected stderr.

```
std::cerr << "SYNC: " << QString(message).toStdString().data() << std::endl;
```

This synchronization framework opens up a possibility for future development of master-backup synchronization.

3.3 Deployment in the cluster

The master process, as is, is capable to run standalone without any cluster. To deploy it in the cluster setup, we need to make two wrapper scripts that allow the master to run correctly in two instances, one master and one backup. These scripts are written in Python. They are directly called by the cluster's rgmanager and mediate it's requests to the master process.

3.3.1 Master control

This script runs the master in the master mode. It is a part of the `master_ip` service (listing 3.3). This scripts detects the commands from the rgmanager (start, stop, status) and redirects them to the master. The start command is enriched by the command `master` which switches the master to the fully working state. The status command is more interesting. Rgmanager regularly call `status` on all scripts it manages. It is it's way how to detect failed service. It checks the status of the master daemon. If the master daemon is running, no action is required. However if the master crashed for any reason, the master control checks for the number of living nodes in the cluster. If there are more than one node alive, it hangs the current node and lets the cluster to switch the master to the second, backup node and fence the local node. If there is only one node available, it exists with failure return code and lets the rgmanager to handle the restart of the master process on the current node. The hanging of the node is issued by a system call: `subprocess.call('echo c > /proc/sysrq-trigger', shell=True)`

```python
def status():
    ret = subprocess.call([master_path, 'status'])
    if ret == 0:
        sys.exit(ret)
    else:
        if two_nodes():
            kill_node()
        else:
            sys.exit(ret) # let the rgmanager handle the restart
```

Listing 3.13: Master control script; status check.

The detection of how many nodes are alive uses an utility called `clustat`. It is an utility that displays current cluster status. It has an option to print the status in the `XML` format, which is easier for computers to process. An example of `clustat -x` output is in the listing 3.15. We use xpath language to navigate through the `XML`. We take an advantage of this in the function `two_nodes`:

```python
from subprocess import Popen, PIPE
def two_nodes():
    try:
        p = subprocess.Popen(['clustat', '-x'])
        p = Popen(['clustat', '-x'], stdout=PIPE, stderr=PIPE)
        output, err = p.communicate()
    except:
        output = None
    if output:
        try:
            root = etree.fromstring(output)
            # at least two nodes are alive
            return len(root.xpath('//node[@state="1"]')) >= 2
        except:
            return False
    else:
        return False
```

Listing 3.14: Two_nodes function. It determines if there are two or more nodes alive in the cluster.

```xml
<?xml version="1.0"?>
<clustat version="4.1.1">
 <cluster name="master_cl" id="18870" generation="424"/>
 <quorum quorate="1" groupmember="1"/>
 <nodes>
    <node name="vm1.dp.jt" state="1" local="1" estranged="0"
    rgmanager="1" rgmanager_master="0" qdisk="0" nodeid="0x00000001"/>
    <node name="vm2.dp.jt" state="0" local="0" estranged="0"
    rgmanager="0" rgmanager_master="0" qdisk="0" nodeid="0x00000002"/>
 </nodes>
 <groups>
    <group name="service:cludaemon_service_1" state="112" state_str="started"
           flags="0" flags_str="" owner="vm1.dp.jt" last_owner="none"/>
    <group name="service:cludaemon_service_2" state="110" state_str="stopped"
           flags="0" flags_str="" owner="none" last_owner="none"/>
    <group name="service:ip_service" state="112" state_str="started"
           flags="0" flags_str="" owner="vm1.dp.jt" last_owner="none"/>
 </groups>
</clustat>
```

Listing 3.15: An example output of clustat -x.

3.3.2 Master Backup

This script is similar to the previous one, but it handles the backup mode of the master. It takes care of the master daemon only on the node that the Master control script is **not** running, otherwise they would interfere. The detection of the location uses again the `clustat` (listing 3.15). The function `master_is_local` (listing 3.16) parses the `XML` output and finds the owner (the node name) of the service `master_ip` that controls the Master control script. Then it checks whether the node is local or not. If the master control currently runs on local node, the master backup scripts does nothing and reports to the rgmanager that everything is working. If the master control runs on the other node, the master backup manages the backup master daemon as the rgmanager desires (redirects start, stop, status calls).

```python
def master_is_local():
  try:
    p = Popen(['clustat', '-x'], stdout=PIPE, stderr=PIPE)
    output, err = p.communicate()
```

```
except:
  output = None
if output:
  try:
    root = etree.fromstring(output)
    # get master node name (mnn)
    mnn = root.xpath('//group[@name="service:master_ip"]')[0].attrib['owner']
    return int(root.xpath('//node[@name="%s"]' % mnn)[0].attrib['local']) == 1
  except:
    return False
else:
  return False
```

Listing 3.16: A function that detects whether the master_ip service is local or not.

3.4 Master Cluster Monitor

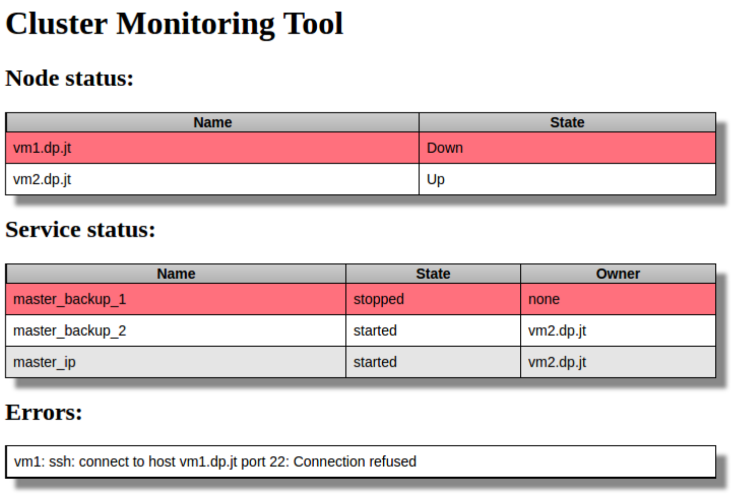
Master Cluster Monitor is a web-based tool that I have custom-made for this cluster. It is able to display the current status of the cluster. It shows an overview of nodes and services. It informs about the state of nodes, about the state of services and where the services are currently located. The layout is a simple table (figure 3.2). It is also able to inform about various error it encountered during obtaining the cluster info. It is written in `Django` using Python. The web server should run somewhere outside the cluster. It uses SSH (listing 3.17) to connect to the nodes of the cluster to get the output of the `clustat -x` command. It first tries the first node, if it fails, it tries the second. If the cluster is running, at least one should be accessible.

```
def ssh_return_output(user, host, cmd):
    command = 'ssh %s@%s %s' % (user, host, cmd)
    p = Popen(command.split(), stdout=PIPE, stderr=PIPE)
    output, err = p.communicate()
    returncode = p.returncode
    return returncode, output, err
```

Listing 3.17: A function that connects to remote hosts. It executes a command and returns it's output.

It utilises functions `get_services` (listing 3.18) and `get_nodes` to parse the XML output obtained from the nodes. Based on this informations, an HTML template is rendered and presented to the user.

```
def get_services(xml):
```



Cluster Monitoring Tool

Node status:

Name	State
vm1.dp.jt	Down
vm2.dp.jt	Up

Service status:

Name	State	Owner
master_backup_1	stopped	none
master_backup_2	started	vm2.dp.jt
master_ip	started	vm2.dp.jt

Errors:

vm1: ssh: connect to host vm1.dp.jt port 22: Connection refused

Figure 3.2: A user interface of the Cluster Monitoring Tool.

```
    try:
        root = etree.fromstring(xml)
        groups = root.xpath('//group')
        services_list = []
        for g in groups:
            service = Service()
            service.name = remove_prefix(g.attrib['name'], 'service:')
            service.state = g.attrib['state_str']
            service.owner = g.attrib['owner']
            service.attention = None if service.state == 'started' else True
            services_list.append(service)
        return services_list
    except:
        return None
```

Listing 3.18: This parses the output of clustat -x command and extracts service info

Chapter 4

Tests

The aim of this chapter is to verify the correct behaviour of the newly developed cluster system. Three possible software or hardware failure scenarios will be explored and tested. We will be watching log outputs of the cluster software, the output of the Cluster Monitoring Tool, and an overview of services detected by the `DIALOG Communication GUI`. The test setup is such that the host computer is running the software for monitoring (CMT), communication (`DIALOG Communication Control Server` and `DIALOG Communication GUI`), and `VirtualBox`. There are two virtualized SLC 6.7 nodes which are running the cluster software.

4.1 Normal, uninterrupted operation

Starting with both nodes running and providing all services, we can check all tools to see how normal operation looks like.

4.1.1 Syslog output of the cluster software

When everything works as it should be, the `rgmanager` periodically checks the status of services. In this case, the `master_ip` cluster service runs on the first node. Currently the status check interval is set to 10s on both nodes. The `rgmanager` logs into the syslog. A short snippet of syslog located at `/var/log/messages` is shown in listings bellow.

```
May 8 19:29:46 vm1 rgmanager[5312]: [script] Executing /tmp/master_backup.py status
May 8 19:29:46 vm1 rgmanager[5450]: [script] Executing /tmp/master_control.py status
May 8 19:29:56 vm1 rgmanager[5516]: [script] Executing /tmp/master_backup.py status
May 8 19:29:57 vm1 rgmanager[5683]: [script] Executing /tmp/master_control.py status
```

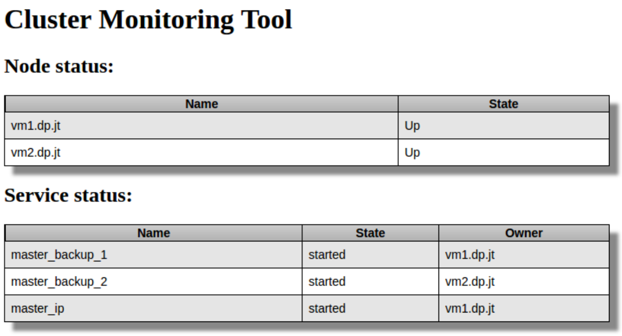
Listing 4.1: Syslog of node one.

```
May 8 19:33:31 vm2 rgmanager[25273]: [script] Executing /tmp/master_backup.py status
May 8 19:33:41 vm2 rgmanager[25321]: [script] Executing /tmp/master_backup.py status
```

Listing 4.2: Syslog of node two.

4.1.2 Cluster Monitoring Tool

The `Cluster Monitoring Tool` (CMT) reports that both cluster nodes are up and running and that all clustered services are provided.



Cluster Monitoring Tool

Node status:

Name	State
vm1.dp.jt	Up
vm2.dp.jt	Up

Service status:

Name	State	Owner
master_backup_1	started	vm1.dp.jt
master_backup_2	started	vm2.dp.jt
master_ip	started	vm1.dp.jt

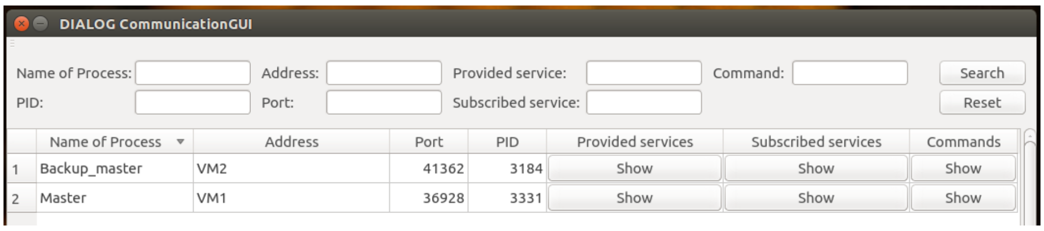
Figure 4.1: A normal operation of the cluster displayed in the CMT.

4.1.3 DIALOG Communication GUI

The `DIALOG Communication GUI` shows two processes running, the DAQ Master on the first node and the backup on the second node. (Figure 4.2). We can further check that the processes are providing all services and that the backup process is subscribed to the `MASTER_SYNC` service and that SYNC data is being sent (Figure 4.3).

4.2 Test Case One - Two nodes service crash

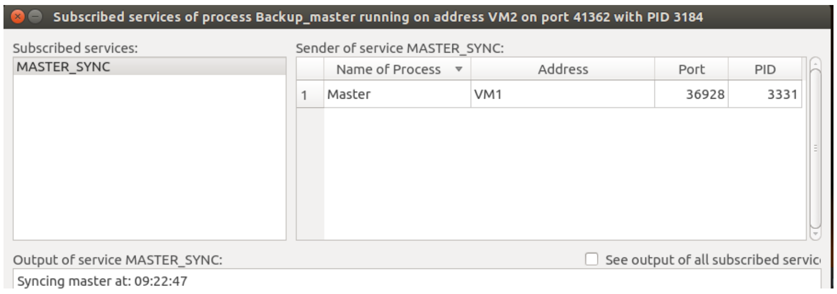
The first test simulates a crash of the master process. We can manually issue `/tmp/master_control.py stop` (in the future, change the path to the


DIALOG CommunicationGUI

Name of Process: Address: Provided service: Command: Search
PID: Port: Subscribed service: Reset

	Name of Process	Address	Port	PID	Provided services	Subscribed services	Commands
1	Backup_master	VM2	41362	3184	Show	Show	Show
2	Master	VM1	36928	3331	Show	Show	Show

Figure 4.2: A normal operation of the cluster displayed in the DIALOG Communication GUI.


Subscribed services of process Backup_master running on address VM2 on port 41362 with PID 3184

Subscribed services:
MASTER_SYNC

Sender of service MASTER_SYNC:

	Name of Process	Address	Port	PID
1	Master	VM1	36928	3331

Output of service MASTER_SYNC: ☐ See output of all subscribed servic

```
Syncing master at: 09:22:47
```

Figure 4.3: A backup process is subscribed to the `MASTER_SYNC` service. The output of the service is shown in the lower part of the image.

current valid path to the control script). An alternative is to call `killall master` which sends a kill signal to all processed with the name `master`. Both ways are equal, it stops the master process. Within 10 seconds (shortest possible status check interval), the `rgmanager` checks for the status of the master process. It will return a non-zero return code as the master is not running anymore.

As mentioned earlier (listing 3.13), the `master_control.py` script decides what to do by determining how many nodes are alive. In this setup both are alive and therefore it decides to kill the current (the first) node. The last status check is below (4.3).

```
May 8 08:05:58 vm1 rgmanager[8494]: [script] Executing /tmp/master_control.py status
```

Listing 4.3: The first status check after the master has been crashed.

The second node detected that the first node is unresponsive in 9 seconds, when the `Totem` declared the first node dead. The next phase, fencing, starts two seconds after that. The fence agent successfully fences the first node, it took around three seconds. The survived node takes over the clustered `master_ip` service. This process requires first to add a specified IP address to a network interface (it consumes four seconds) and then starting the full master process. Because the backup process has already been running on the second node, the switching to the master mode is only done by sending a `SIGUSR1` signal. The time it took from the moment the first node crashed to the successful switch to the master mode on the second node, was roughly 19 seconds. If this happened in the real deployment it would mean only roughly one or two spills would be lost. The output from the syslog of the second node is in the listing 4.4.

```
May 8 08:06:07 vm2 corosync[1322]: [TOTEM ] A processor failed, forming new configuration.
May 8 08:06:09 vm2 corosync[1322]: [QUORUM] Members[1]: 2
May 8 08:06:09 vm2 corosync[1322]: [TOTEM ] A processor joined or left the membership
                                    and a new membership was formed.
May 8 08:06:09 vm2 kernel: dlm: closing connection to node 1
May 8 08:06:09 vm2 corosync[1322]: [CPG ] chosen downlist: sender r(0) ip(10.0.0.102) ;
                                    members(old:2 left:1)
May 8 08:06:09 vm2 corosync[1322]: [MAIN ] Completed service synchronization, ready to
                                    provide service.
May 8 08:06:09 vm2 rgmanager[1946]: State change: vm1.dp.jt DOWN
May 8 08:06:09 vm2 fenced[1386]: fencing node vm1.dp.jt
May 8 08:06:12 vm2 fenced[1386]: fence vm1.dp.jt success
May 8 08:06:12 vm2 rgmanager[4135]: [script] Executing /tmp/master_backup.py status
May 8 08:06:12 vm2 rgmanager[1946]: Marking service:master_backup_1 as stopped:
                                    Restricted domain unavailable
May 8 08:06:12 vm2 rgmanager[1946]: Taking over service service:master_ip from down
                                    member vm1.dp.jt
May 8 08:06:12 vm2 rgmanager[4225]: [ip] Adding IPv4 address 10.0.0.10/24 to eth1
May 8 08:06:16 vm2 rgmanager[4314]: [script] Executing /tmp/master_control.py start
May 8 08:06:17 vm2 [3163]: Switching to master mode. SIGUSR1 received.
May 8 08:06:17 vm2 rgmanager[1946]: Service service:master_ip started
```

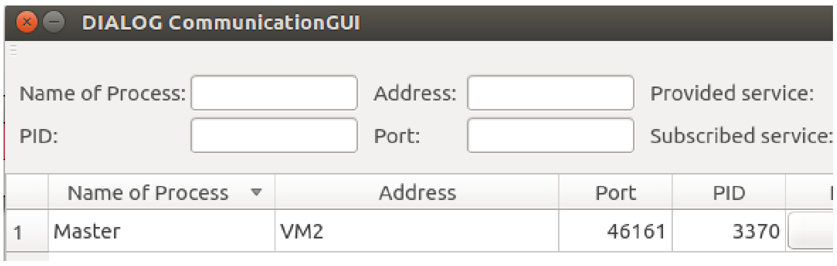
Listing 4.4: The syslog output of the second node. Service relocation.

The fencing agent rebooted the first node. The node then rejoined the cluster and started providing `master_backup_1` service approximately after 34 seconds after being successfully fenced, see listing from the second node 4.5.

```
May 8 08:06:34 vm2 corosync[1322]: [TOTEM ] A processor joined or left the membership
                                    and a new membership was formed.
May 8 08:06:34 vm2 corosync[1322]: [QUORUM] Members[2]: 1 2
May 8 08:06:34 vm2 corosync[1322]: [QUORUM] Members[2]: 1 2
May 8 08:06:34 vm2 corosync[1322]: [CPG ] chosen downlist: sender r(0) ip(10.0.0.101) ;
                                    members(old:1 left:0)
May 8 08:06:34 vm2 corosync[1322]: [MAIN ] Completed service synchronization,
                                    ready to provide service.
May 8 08:06:35 vm2 rgmanager[4402]: [script] Executing /tmp/master_control.py status
May 8 08:06:41 vm2 kernel: dlm: got connection from 1
May 8 08:06:45 vm2 rgmanager[4530]: [script] Executing /tmp/master_backup.py status
May 8 08:06:46 vm2 rgmanager[4672]: [script] Executing /tmp/master_control.py status
May 8 08:06:46 vm2 rgmanager[1946]: State change: vm1.dp.jt UP
```

Listing 4.5: The syslog output of the second node. The first node joining back.

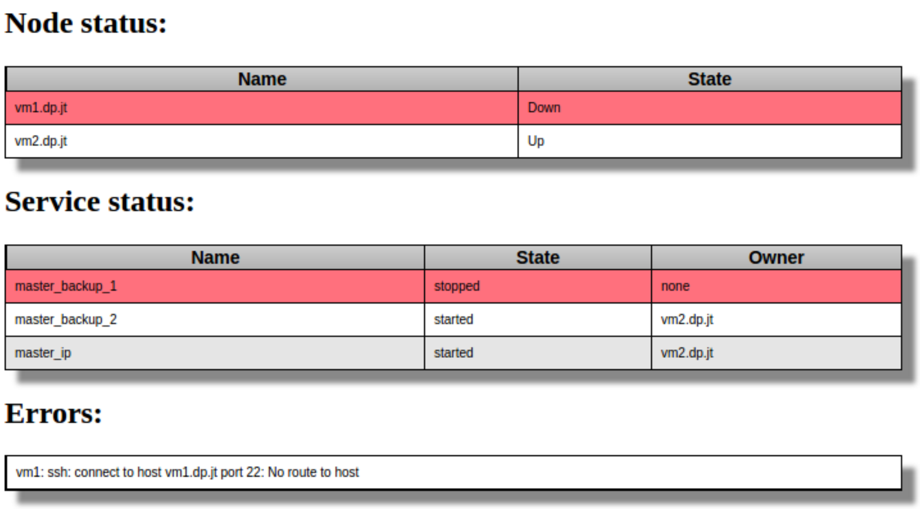
The `DIALOG Communication GUI` follows the progress of the cluster. At first the `Master` process disappears from the first node. When the second node takes over the clustered `master_ip` service, the `Backup_master` process on the second node is replaced by the new `Master` process on the new host. This state is shown on the figure 4.4. When the first node rejoins the cluster, a new `Backup_master` process appears with the address of the first node. The two processes effectively switched their owners.



	Name of Process	Address	Port	PID
1	Master	VM2	46161	3370

Figure 4.4: The Master process has been relocated to the second node.

The output of the Cluster Monitoring Tool from the moment before the first node rejoins the cluster shows (Figure 4.5) the correct state and displays errors it encountered during retrieving the cluster status information.



Node status:

Name	State
vm1.dp.jt	Down
vm2.dp.jt	Up

Service status:

Name	State	Owner
master_backup_1	stopped	none
master_backup_2	started	vm2.dp.jt
master_ip	started	vm2.dp.jt

Errors:

```
vm1: ssh: connect to host vm1.dp.jt port 22: No route to host
```

Figure 4.5: The first node is rebooting.

One thing should be mentioned. According to the `cluster.conf` (listing 3.2), the first node is always fenced immediately while the fence agent always waits 10 seconds before fencing the second node. In a reversed case scenario (if the two nodes swapped their roles), the whole process would take 10 more seconds because of the fencing delay.

4.3 Test Case Two - Network Failure

This test is trying to simulate a network failure. It should test whether the current cluster setup is resilient to the split-brain and fence-races situation. Fence-race situation is a situation in which both nodes try to fence each other in order to bring the second node to the known state. It could end-up with both nodes fenced - turning the cluster off. In the real world deployment, nodes should have multiple network interfaces, each for different type of communication (inter-cluster communication - TOTEM, corosync, connection to fence devices, and more).

However in this testing setup we have only one network interface that is used for both tasks. In order to block communication between the nodes without breaking communication to the fence device, `iptables` comes in handy. The task is to block the `corosync` communication which listens by default on ports 5404 and 5405. It can be found using `netstat` command (listing 4.6):

```
[root@vm1 ~]$ netstat -lptnu | grep corosync
udp  0  0  10.0.0.101:5404  0.0.0.0:*    1727/corosync
udp  0  0  10.0.0.101:5405  0.0.0.0:*    1727/corosync
udp  0  0  239.192.73.0:5405  0.0.0.0:*    1727/corosync
```

Listing 4.6: The output of the netstat utility. Detecting the ports corosync uses.

The default `iptables` configuration blocks this kind of connection. So turning the `iptables` on (`service iptables start`) on both simultaneously creates exactly the kind of situation we want to test. Both nodes are alive, they can't communicate with each other, and they can communicate with the fence device (the communication is done over the ssh). The only thing they can do is to assure that they know the other node is in a known state, i.e. fenced. Let's take a look at the output from the syslog on the first (listing 4.7) and the second node (listing 4.8).

```
May 8 23:16:31 vm1 kernel: ip_tables: (C) 2000-2006 Netfilter Core Team
May 8 23:16:39 vm1 rgmanager[23947]: [script] Executing /tmp/master_backup.py status
May 8 23:16:41 vm1 corosync[1312]: [TOTEM ] A processor failed, forming new configuration.
May 8 23:16:43 vm1 corosync[1312]: [QUORUM] Members[1]: 1
May 8 23:16:43 vm1 corosync[1312]: [TOTEM ] A processor joined or left the membership
                                              and a new membership was formed.
```

```
May 8 23:16:43 vm1 kernel: dlm: closing connection to node 2
May 8 23:16:43 vm1 corosync[1312]: [CPG ] chosen downlist: sender r(0) ip(10.0.0.101) ;
                                    members(old:2 left:1)
May 8 23:16:43 vm1 corosync[1312]: [MAIN ] Completed service synchronization,
                                    ready to provide service.
May 8 23:16:43 vm1 rgmanager[22326]: State change: vm2.dp.jt DOWN
May 8 23:16:43 vm1 fenced[1379]: fencing node vm2.dp.jt
```

Listing 4.7: Syslog of the first node after turning iptables on.

```
May 8 23:16:31 vm2 kernel: ip_tables: (C) 2000-2006 Netfilter Core Team
May 8 23:16:38 vm2 rgmanager[18398]: [script] Executing /tmp/master_control.py status
May 8 23:16:41 vm2 corosync[1323]: [TOTEM ] A processor failed, forming new configuration.
May 8 23:16:43 vm2 corosync[1323]: [QUORUM] Members[1]: 2
May 8 23:16:43 vm2 corosync[1323]: [TOTEM ] A processor joined or left the membership
                                    and a new membership was formed.
May 8 23:16:43 vm2 kernel: dlm: closing connection to node 1
May 8 23:16:43 vm2 corosync[1323]: [CPG ] chosen downlist: sender r(0) ip(10.0.0.102) ;
                                    members(old:2 left:1)
May 8 23:16:43 vm2 corosync[1323]: [MAIN ] Completed service synchronization,
                                    ready to provide service.
May 8 23:16:43 vm2 rgmanager[15917]: State change: vm1.dp.jt DOWN
May 8 23:16:43 vm2 fenced[1382]: fencing node vm1.dp.jt
May 8 23:16:46 vm2 fenced[1382]: fence vm1.dp.jt success
May 8 23:16:46 vm2 rgmanager[15917]: Marking service:master_backup_1 as stopped:
                                    Restricted domain unavailable
May 8 23:16:48 vm2 rgmanager[18467]: [script] Executing /tmp/master_backup.py status
```

Listing 4.8: Syslog of the first node after turning iptables on.

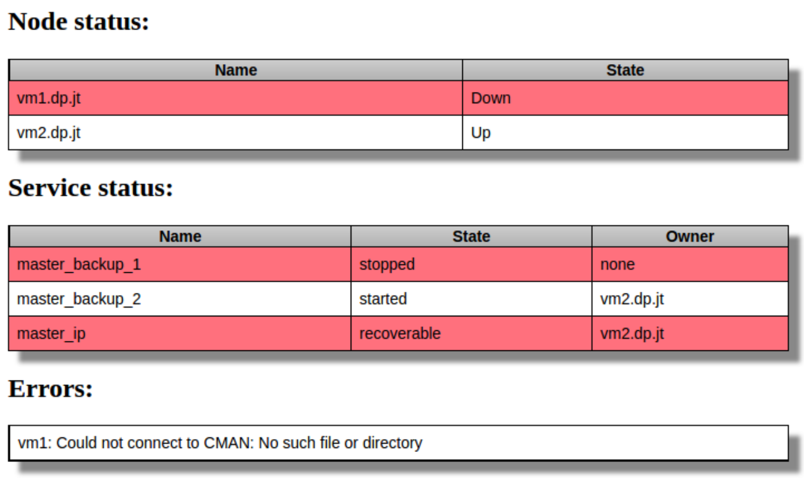
We can clearly see that 10 seconds after turning the `iptables` on, both nodes declared the other node dead. Then they started fencing the (from their own perspective) lost node at exactly the same time. The second node survived and successfully fenced the first node. No fence-race nor split-brain has occurred. It was caused by the `delay` attribute in the fence device definition in `cluster.conf` file. It told the first node to wait before fencing. Without the delay, the result would be undetermined, it could have finished with both nodes dead, with one alive, or with both alive.

4.4 Test Case Three - Single node service crash

The last from the series of tests will look into the behaviour of the cluster when only one node is alive. The setup of fail-over domains ensures that the clustered `master_ip` service will always run on the node that is alive. It means that the master process will be running as well. We can now test how long it takes to restart the master process should it crash. To create this situation, we manually turn off the first node. The resulting (initial) setup will be the same as described by the figure 4.5.

Now we can crash the master process. To do so, we can issue following command, which immediately kills the process and loggs it into the syslog for better track of time.

```
kill −9 ‘cat /tmp/master_daemon.pid‘ && logger Master manually killed.
```



Node status:

Name	State
vm1.dp.jt	Down
vm2.dp.jt	Up

Service status:

Name	State	Owner
master_backup_1	stopped	none
master_backup_2	started	vm2.dp.jt
master_ip	recoverable	vm2.dp.jt

Errors:

vm1: Could not connect to CMAN: No such file or directory

Figure 4.6: The recovery of the `master_ip` service.

The cluster found out that the `master` process is dead during the first status check after it was killed. We can now clearly see that the `master_control.py` script behaves differently from the first Test Case. Now it correctly detected that there is only one node alive and therefore it did not stop this node and it passed the control to the rgmanager. Rgmanager is set to try to restart this failed service. The `master_ip` service is defined as a resource tree with the `IP` resource as the parent and the `master_control.py` script as the child. This ensures that the master process is started only and only in the case that the predefined IP address is active on the node. The listing 4.9 contains this whole process, removing and adding IP address is recorded as the whole service is begin restarted. The whole process of recovery takes approximately 26 seconds. The figure 4.6 catches the moment of the recovery of the `master_ip` service.

```
May 9 01:07:50 vm2 rgmanager[5212]: [script] Executing /tmp/master_control.py status
May 9 01:07:52 vm2 root: Master manually killed.
May 9 01:08:00 vm2 rgmanager[5308]: [script] Executing /tmp/master_backup.py status
May 9 01:08:00 vm2 rgmanager[5451]: [script] Executing /tmp/master_control.py status
May 9 01:08:01 vm2 rgmanager[5484]: [script] script:master_control:
                                    status of /tmp/master_control.py failed (returned 1)
May 9 01:08:01 vm2 rgmanager[3471]: status on script "master_control" returned 1 (generic err)
May 9 01:08:01 vm2 rgmanager[3471]: Stopping service service:master_ip
May 9 01:08:01 vm2 rgmanager[5524]: [script] Executing /tmp/master_control.py stop
May 9 01:08:01 vm2 rgmanager[5585]: [ip] Removing IPv4 address 10.0.0.10/24 from eth1
May 9 01:08:10 vm2 rgmanager[5628]: [script] Executing /tmp/master_backup.py status
May 9 01:08:11 vm2 rgmanager[3471]: Service service:master_ip is recovering
May 9 01:08:11 vm2 rgmanager[3471]: Recovering failed service service:master_ip
May 9 01:08:12 vm2 rgmanager[5721]: [ip] Adding IPv4 address 10.0.0.10/24 to eth1
May 9 01:08:15 vm2 rgmanager[5804]: [script] Executing /tmp/master_control.py start
May 9 01:08:16 vm2 V[5829]: Switching to master mode. Signal SIGUSR1 received.
May 9 01:08:16 vm2 rgmanager[3471]: Service service:master_ip started
```

Listing 4.9: A single node service recovery test.

Conclusion

This diploma thesis was focused on increasing the availability of the main process of the DAQ system (the master process), creating a backup system, and on finding a way of synchronization of the main and the backup processes.

The higher availability and the backup process have been implemented using a high-availability cluster software. It produced a system that is both hardware and software fault tolerant. With the deployment of the master process in the cluster, new requirements to this process have arisen. All of them have been successfully satisfied and implemented.

During the development process a need for a cluster monitoring tool appeared. As a result, a new web-based tool for cluster monitoring has been created. It can now be deployed in the control room to help the crew to access up-to-date information about the cluster status and services it provides.

A draft of how to synchronize arbitrary data between the main master process and it's backup sibling has been presented. It is up to the future projects to fully utilise the suggested way of synchronization.

The goal of this diploma thesis has been fully fulfilled.

Bibliography

[1] **J. Tomsa:** *Monitoring tools for the data acquisition system of the COMPASS experiment at CERN*
Prague, Czech Technical University in Prague, June 2014

[2] **http://cern.ch/**
European Organization for Nuclear Research
[online] cited in May 2016

[3] **Adolph Ch. et al. (The COMPASS Collaboration):** *COMPASS-II Proposal*
CERN-SPSC-2010-014; SPSC-P-340, May 2010.

[4] **M Bodlak, V Frolov, S Huber, V Jary, I Konorov, D Levit, J Novy, R Salac, J Tomsa and M Virius**

Monitoring tools of COMPASS experiment at CERN
Int. Conf. Proc. CHEP2015

[5] *M. Bodlak, V. Frolov, V. Jary, S. Hube,r I. Konorov, D. Levit, A. Mann, J. Novy, S. Paul, and M. Virius:*
New data acquisition system for the COMPASS experiment
Topical Workshop on Electronic for Particle Physics, September 2012

[6] *Anticic T. et al.:* **(ALICE DAQ Project): ALICE DAQ and ECS User's Guide)**
CERN, EDMS 616039, January 2006

[7] **M. Bodlák:** *COMPASS DAQ – Database architecture and support utilities.*
Prague, Czech Technical University in Prague, June 2012

[8] **J. Nový:** *COMPASS DAQ - Basic Control System.*
Prague, Czech Technical University in Prague, June 2012

[9] **en.wikipedia.org/wiki/High-availability_cluster**
High-availability, Wikipedia
[online] cited in April 2016

[10] **https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/ Configuration_Example_-_Oracle_HA_on_Cluster_Suite/images/2-node-oracle.png**
Red Hat Documentation
[online] cited in April 2016

[11] **https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Cluster_Suite_Overview/**
Red Hat Cluster Suite, documentation
[online] cited in April 2016

[12] **https://fedorahosted.org/cluster/wiki/RGManager**
Resource Group Manager, documentation
[online] cited in April 2016

[13] **https://alteeve.ca/w/Rgmanager**
Resource Group Manager, wiki
[online] cited in April 2016

[14] **http://www.cplusplus.com/info/description/**
About C++
[online] cited in May 2016

[15] **https://en.wikipedia.org/wiki/C++**
About C++, wiki
[online] cited in May 2016

[16] **https://wiki.qt.io/About_Qt**
About Qt, Qt wiki
[online] cited in May 2016

[17] **https://en.wikipedia.org/wiki/Python_(programming_language)**
Python programming language, wiki
[online] cited in May 2016

[18] **https://en.wikipedia.org/wiki/Django_(web_framework))**
Django, wiki
[online] cited in May 2016

[19] **http://dim.web.cern.ch/**
The DIM website
[online] cited in May 2016

[20] **https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Cluster_Administration/ch-config-cli-CA.html**
RHCS configuration, documentation
[online] cited in May 2016

[21] **https://fedorahosted.org/cluster/wiki/FenceAgentAPI**
Fence agent API
[online] cited in May 2016

[22] **http://www.linfo.org/daemon.html**
Unix Daemon definition
[online] cited in May 2016

[23] **http://lib.ru/UNIXFAQ/unixprogrfaq.txt**
Unix Programming FAQ
[online] cited in May 2016

[24] **http://refspecs.linuxbase.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/iniscrptact.html**
LSB Compliant return codes for daemon.
[online] cited in May 2016

Appendix A

Installation Guide

This appendix is meant to be a guide to install and deploy the software solution presented in this thesis. It can also be used as a quick reference for useful commands and various observations.

The PDF version of this thesis and all source codes are available on the enclosed DVD on the back cover of this thesis.

A.1 Virtual Machines

The software in this thesis has been developed with help of virtual machines. The final product is also meant to be deployed on virtual machines.

A.1.1 VirtualBox

VirtualBox is one of many tools for virtualization. It can downloaded either from the virtualbox site (`https://www.virtualbox.org/wiki/Downloads`) or from repositories.

```
sudo apt-get install virtualbox #debian based distributions
sudo yum install virtualbox #RHEL based distributions
```

A.1.2 Vagrant

Vagrant (`https://www.vagrantup.com/`) is a very useful tool for managing development environments. It is downloadable from the webpage or from repositories. It provides automatized creating of virtual machines, working on top of virtualbox and other providers of virtual machines. It is able to obtain a machine image (so called `box`), provision the machine (install various packages, setup network, etc.), setup ssh access, mount shared folders .... It is not necessary to use, but once properly configured speeds up the initial development. Starting is simple:

```
cd path/to/working/dir
vagrant init [box name] #for example vagrant init indatus/sl65
```

The box used in this thesis was indatus/sl65, downloaded from the box repository https://atlas.hashicorp.com. I slightly upgraded it and repacked it and it is available on the DVD.

`Vagrant init` creates a new vagrant environment, within which there is a `Vagrantfile`. It is the main configuration file for vagrant. It defines which boxes to download, how many machines to manage, setup their names, RAM, CPU, ip addreses, ssh access, etc. See the enclosed DVD for an example of the `Vagrantfile`. The table below shows some daily used commands.

Start all machines	vagrant up
Start specific machine	vagrant up vm1
Stop machines	vagrant halt
SSH connection to machine	vagrant ssh vm1

A.1.3 VirtualBox additions

Once the machines are running, I recommend to install VirtualBox additions, which make the work with virtual machines more pleasant. Go to the menu of the running machine - section `Devices/Insert guest addition CD`.

Then head to the virtual machine. Create a directory to mount the CD to. Try to install the guest additions. It might be possible that it would require to have `gcc` and kernel headers installed.

```
mkdir /mnt/cdrom
mount -t iso9660 -o ro /dev/sr0 /mnt/cdrom
yum install gcc kernel-devel-2.6.32-573
sudo /mnt/cdrom/VBoxLinuxAdditions.run --nox11 #nox11 if you are not using X
```

A.2 Cluster software and node setup

There is a list of software that is required to run the cluster. This is the basics. NTP should be installed in order to have the time of all nodes synchronized. Run the following commands.

```
sudo yum update
sudo yum install cman corosync rgmanager ricci pcs fence-agents ntp
```

There is another list of programs that I find very useful during development, but it is optional.

```
sudo yum install rsync vim screen man
```

Ricci and modclusterd daemons

These two daemons take care of propagating modifications to the cluster and of distributing the `cluster.conf` file. Now setup a password for `ricci`, the same on all nodes.

```
passwd ricci
```

Now setup the `ricci` and `modclusterd` daemon to start automatically on each start of the system:

```
chkconfig ricci on
chkconfig modclusterd on
```

SSH access

Setup a password-less ssh access between cluster nodes and to other computers, if required (for example to the virtual machine host).

```
ssh-keygen #creates a new key pair
ssh-copy-id user@address.of.target #this copies public key for passwordless access
```

IPTables

IPTables is a linux firewall. The network is a key resource for cluster. Therefore we need to configure iptables in such a way that it does not block the cluster communication. Configuring iptables is a difficult and large topic on it's own, so there is only an example of how to allow communication for cman (totem, corosync) which uses UDP/multicast on ports 5404 and 5405.

```
iptables -I INPUT -m state --state NEW -m multiport -p udp -s 10.20.0.0/16
    -d 10.20.0.0/16 --dports 5404,5405 -j ACCEPT
iptables -I INPUT -m addrtype --dst-type MULTICAST -m state --state NEW
    -m multiport -p udp -s 10.20.0.0/16 --dports 5404,5405 -j ACCEPT
```

However I would recommend to temporarily turn iptables off and set them up only when needed and when everything else works properly.

```
chkconfig iptables off
```

SELinux

SELinux is a Security-Enhanced Linux, a kernel security module which provides support for access control. We need to configure SELinux so it allows `fenced` daemon to use network and ssh to connect to fence devices.

```
setsebool -P fenced_can_network_connect 1
setsebool -P fenced_can_ssh 1
```

But again, as with `IPTables`, unless you need SELinux, you might want to turn it off for development and turn it on afterwards.

```
vim /etc/sysconfig/selinux #set it to disabled
```

Now is time to restart your virtual machines.

Cluster configuration

The cluster is configured via the `cluster.conf` XML file. Once a first version of the configuration file is ready, copy it to `/etc/cluster/cluster.conf`. Remember to always increment the `config_version` attribute when pushing the new configuration. Validate the configuration. If the configuration validates, sync the changes to other nodes (or you can copy it manually).

```
ccs_config_validate #validates configuration
ccs_sync   #synchronizes configuration accross the cluster
```

If you have any custom fence agents, those (executable) scripts should be place into the `/usr/sbin/` folder. The naming convention is to prefix the script with `fence_`, for example `fence_virtualbox`.

Hosts

Configure hosts file. It is located in `/etc/hosts`. This file serves for translating address strings to actual ip addresses. When configured properly, verbose names (like vm1.dp.jt) can be used system-wide instead of plain ip addresses (10.0.0.101). Even in cluster configuration files. Below there is a sample configuration of the `/etc/hosts` file.

```
127.0.0.1 vm1 localhost
#NODE 1
10.0.0.101 vm1 vm1.dp vm1.dp.jt
#NODE 2
10.0.0.102 vm2 vm2.dp vm2.dp.jt
#VBOXHOST
10.0.0.1 vboxhost vboxhost.dp vboxhost.dp.jt
#MASTER ADDRESS
10.0.0.10 master master.dp master.dp.jt
```

CMAN, Rgmanager

When the configuration file is ready and verified, it is time to start the cluster software.

```
#on every node
service cman start
```

This groups all nodes together to form a cluster. The status of the cluster can be checked with the `clustat` command. It produces an output similar to this:

```
Cluster Status for master_cl @ Mon May 9 05:22:47 2016
Member Status: Quorate

Member Name  ID  Status
------ ---- --- ------
vm1.dp.jt  1  Online, Local
vm2.dp.jt  2  Online
```

The cman only creates a skeleton on which resources can be deployed. In order to start providing resources, rgmanages must be started:

```
service rgmanager start
```

This starts services according to the `cluster.conf` file. The output of the `clustat` is enriched by the info about services:

```
Service Name Owner (Last) State
------- ---- ----- ------ -----
service:master_backup_1 vm1.dp.jt started
service:master_backup_2 vm2.dp.jt started
```

```
service:master_ip vm2.dp.jt started
```

It is very useful to watch the syslog. It helps a lot during development. The cluster software uses it extensively to inform about everything it does.

```
tail -n50 -f /var/log/messages
```

For development purposes it is advisable to start `cman` and `rgmanager` manually. But when it comes to deployment, it is necessary to make these services autostart on boot.

```
chkconfig cman on
chkconfig rgmanager on
```

A.3 The master process

A.3.1 The master

Full source codes for the master process are available on the enclosed DVD or on an online git repository (after requesting for access). The git address is `https://gitlab.cern.ch/COMPASS_RCCARS/compass-rccars-daq`.

Because the project is written using Qt, it is therefore necessary to install Qt libraries. Installing `QtCreator` from repositories should be sufficient, but if the node is not going to be used for development, the qt core should suffice:

```
yum install qt5-qtbase-devel qt5-qtbase-gui qt5-qtbase-mysql
```

For building the applications, load the project into the `QtCreator`, configure, and build. There are several components of the DAQ Software that are required to compile the master process.

- DIM library - the package is on the enclosed DVD; it is downloadable from `https://dim.web.cern.ch/dim/`
- TransportProtocol - a part of the DAQ Software package; handles coding and decoding of internal messages
- DIALOG Communication library
- Database library - a custom made library for the DAQ Software

The DIM and DIALOG Communication libraries are used as shared objects. When deploying the master, it is necessary to make symbolic links to `/usr/lib64` or to setup correctly the `LD_LIBRARY_PATH` environment variable. The master requires also these additional environment variables to be set:

- MasterAdress
- MasterPort
- DIALOG_CONTROL_SERVER_ADDRESS
- DIALOG_CONTROL_SERVER_PORT
- DP_USER
- DB_PASSWD

This is valid for starting the master manually. However when deploying the master as a clustered service, it is unable to read any user defined environment variables (setting them to `/etc/environment` or `/etc/profile` does not work either). The only way is to hardcode these values or to create configuration file which the master is going to read from. The default location of the master pidfile is `/tmp/master_daemon.pid`. This can be also adjusted. The master location should be /online/compass-rccars-daq/compass-rccars-daq-master/master.

A.3.2 Cluster wrappers for the master

The `master_backup.py` and `master_control.py` scripts don't have any special dependences. They are simple python scripts, so the only requirements are python 2 and a lxml python library. This library can be obtained using pip:

```
pip install lxml
```

The location of these scripts is up to the user, but their current location must reflect in the `cluster.conf` file. Before using, they should be edited to adjust the `master_path`.

A.4 Cluster Monitoring Tool

This monitoring tool is developed in Django using Python. It is recommended to install this application into a virtualenv.

```
cd /path/to/cmt
virtualenv clu_mon
cd clu_mon
source bin/activate
pip install django==1.9
pip install lxml
cp /path/to/cmt-source /path/to/cmt/clu_mon/
cd clu_mon
./manage.py runserver 0.0.0.0:[port]
```

After this sequence, a web server should be running and the application should be available at port [port] on the machine it started on. It is however recommended not to use this django development webserver for deployment, apache2 or nginx should be used instead.