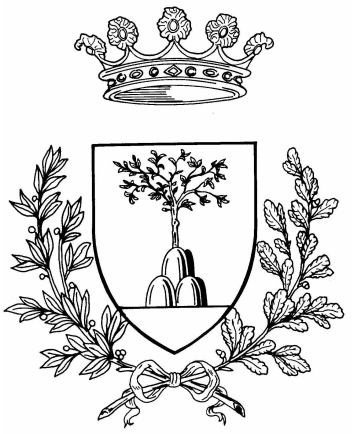
Development of the acquisition system and the control environment for the experiment KM3NeT



Matteo Favaro

Department of Mathematics and Computer Science

University of Ferrara

This dissertation is submitted for the degree of

Doctor of Philosophy

Co Tutor:

Tommaso Chiarusi

Francesco Giacomini

Tutor:

Luca Tomassetti

I would like to dedicate this thesis to:

my loving parents, that have supported me during these years,

Chiara, who has been one of the true friends in my life,

and to the persons that are making me happy. She understands me as nobody has ever done. . .

Thank to all of you. Without your support, I never reached this achievement.

Matteo Favaro

Acknowledgements

I would like to acknowledge: Francesco Giacomini for helping me during this period, Tommaso Chiarusi for having kept the workgroup close, Matteo Manzali for the friendship and the help on this work and last but not least Carmelo Pellegrino for giving me all his patience and encouragements. These persons will have a important place in my heart and life forever.

Thank you sincerely. . .

Matteo Favaro

Abstract

In this thesis the Trigger and Data Acquisition System (TriDAS) for the KM3NeT-Italy experiment is presented. The various elements that compose the TriDAS are explained in detail, together with the description of the software tools used for their design and implementation. The control system of the data acquisition is one of the key features of the whole system and it is the core of this work. To develop it, a general approach has been used, with the aim for possibly reusing most of the product within other projects. The adopted solution improves the user interface, exposing web-service API for remote steering and control; it also allows the connections by multiple users at the same time, managing different privileges for many user roles. Finally, the results of some tests are shown, demonstrating that the new design of control system work in a reliable way. Moreover, the test probes the TriDAS performances under realistic conditions.

Table of contents

List of figures

List of tables

Introduction

The INFN's project KM3NeT-Italy [1], supported with Italian PON (National Operative Programs) fundings, consists of 8 vertical structures, called Towers, instrumented with a total number of 672 Optical Modules (OMs) and will be deployed 3500 m deep in the Ionian Sea, at about 80 km from the Sicilian coast [2][3]. A Tower is made of 14 horizontal bars, piled up one by one with 90° heading difference. Each bar hosts six OMs. In order to reduce the complexity of the underwater detector, the *all-data-to-shore* approach is followedAt the shore station a Trigger and Data Acquisition System collects, processes and filters the data streams from all the Towers, saving interesting data to a permanent storage for subsequent analysis.[4]

The system must sustain the real data generated from the neutrinos added to the optical background generated by the ^{40}K decays and bioluminescence. The throughput can grow up to 30 Gbps, with such large throughput strong constraints are required by the TriDAS performances and by the related networking architecture.

Chapter 1 briefly describes the experiment and discusses the implications and the challenges for its acquisition system. In chapter 2, I will present how the TriDAS is built and which are its components. Chapter 3 shows all the work that I have developed for the TriDAS control system. In chapter 4 the result of a

realistic simulation will be presented in order to demostrate that the system is ready for the real data acquisition.

Chapter 1

The KM3NeT Experiment

The primary aim of the KM3NeT project [5] is the detection of high-energy neutrinos from the cosmos. Following the construction and operation of the ANTARES neutrino telescope, the completion of the EU funded Design Study and Preparatory Phase Study, and the acquisition of substantial funds (about 20% of the envisaged total budget), KM3NeT phase-1 was launched in early 2013. The construction has started off-shore Porto Palo di Passero, Italy and Toulon, France. The construction of a neutrino telescope is extremely challenging. In short, a systematic study of cosmic neutrinos requires a massive telescope with a size of several cubic kilometres. A solution to make such a large mass sensitive to neutrinos is to build a three dimensional array of very sensitive light sensors in the sea. Neutrinos can then be detected indirectly through the detection of the Cherenkov light produced by charged particles emerging from a neutrino interaction. The transparency of the water makes it possible to distribute the light sensors in a cost effective way. The absorption length of the water has been measured at the selected sites in the Mediterranean Sea and was found to be about 50 metres (at a wavelength of 470 nm). The angular resolution of such a detector is limited by the lever arm between the light sensors and the

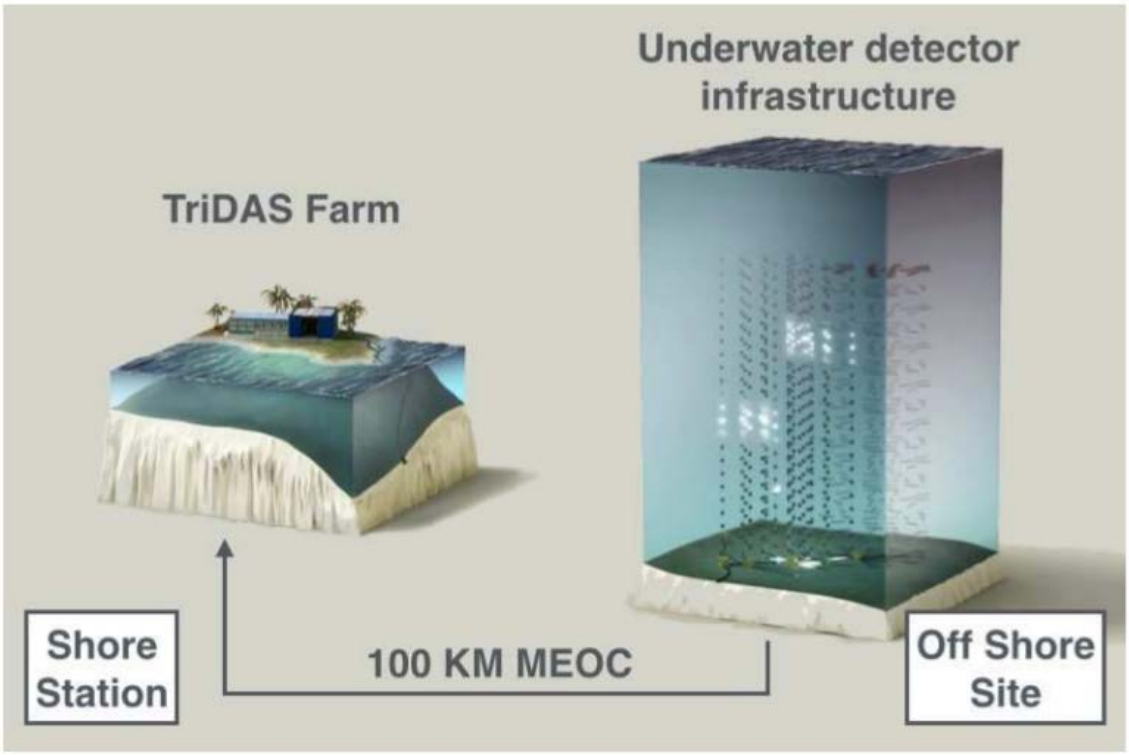


Fig. 1.1 The experimental setup of the neutrino telescope in the KM3NeT-IT site: the grid of optical modules, in the abyssal site, is connected to shore via a mechanical electro-optical cable (MEOC) about 100 km long. On shore the control station hosts the TriDAS computing farm for the on-line data processing.

measurement precision of their positions and the arrival times of the Cherenkov light. The mechanical structure that accommodates the light sensors do not form a static system due to changing sea currents. Hence, their positions must be monitored continuously through acoustic triangulation. Of the three neutrino species that exist in nature, the muon neutrino yields the best angular resolution because the muon that emerges from a neutrino interaction has the longest range. The KM3NeT infrastructure will also host a network of cabled observatories with a wide array of dedicated instruments for oceanographic, geophysical and marine biological research.

1.1 Neutrino Detection Techniques

A number of possible techniques exists for detecting high energy neutrinos from space. The most widely exploited method for the core energy range of interest (10^{11} to 10^{16} eV) is the detection of neutrinos in large volumes of water or ice, using the Cherenkov light from the muons and hadrons produced by neutrino interactions with matter around the detector. The original idea of a neutrino telescope based on the detection of the secondary particles produced in neutrino interactions is attributed to Markov[6] who invoked the concept in the 1950's. Given the need of a kilometrescale detector, only designs incorporating large naturally occurring volumes of water or ice can be viable. A deep seawater telescope has significant advantages over ice and lake-water experiments due to the better optical properties of the medium. Water Cherenkov detectors are the only detectors that have so far observed neutrinos produced beyond the solar system; these were neutrinos with energies of 10^6 to 10^7 eV produced in supernova 1987a. These detectors are much smaller than the proposed KM3NeT neutrino telescope. MACRO was the largest detector using a different technique.

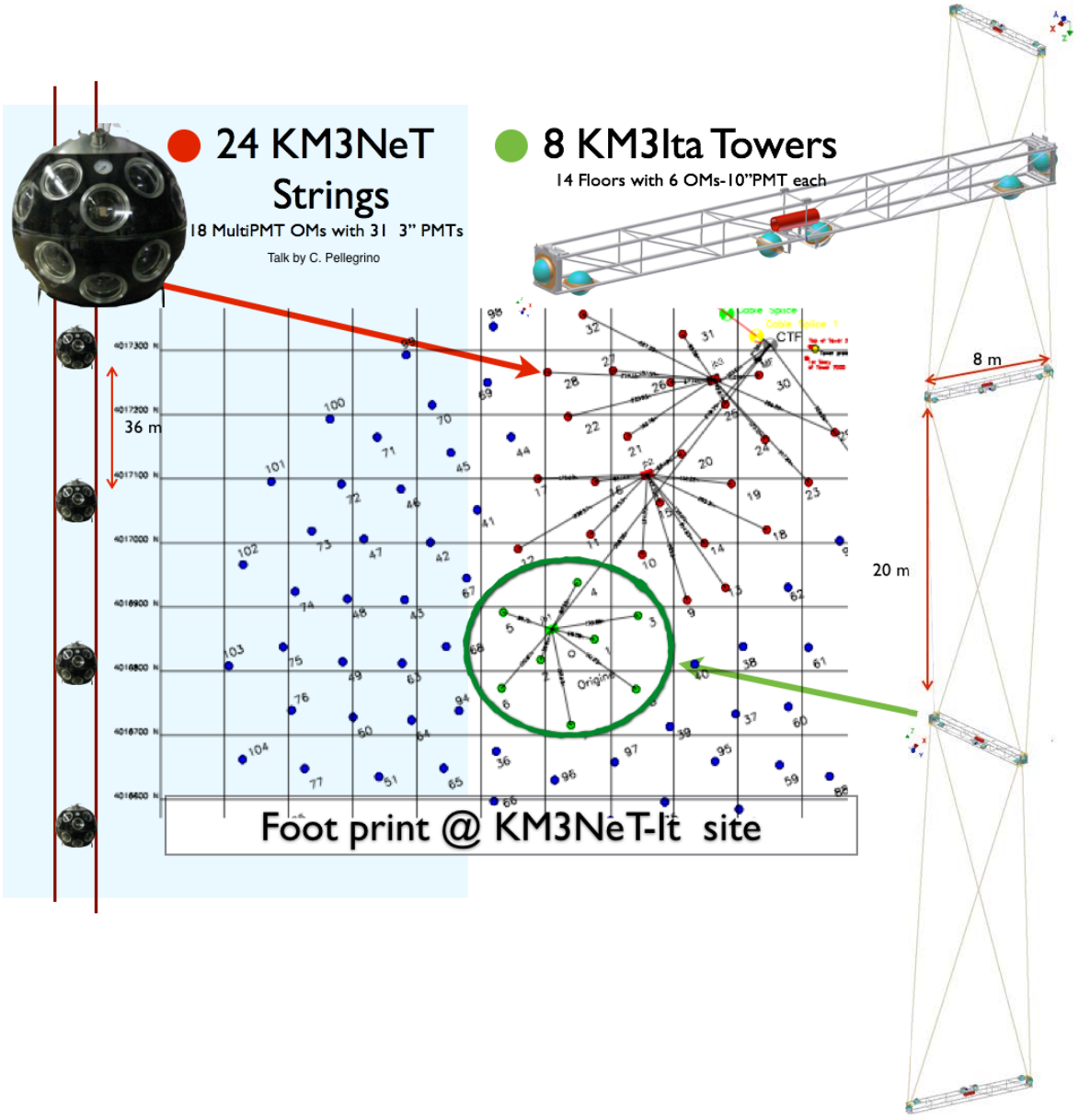


Fig. 1.2 Km3 experiment Foot print and visual illustration of towers and strings, on the right side is illustrated hot a tower is built, with perpendicular floors connected by wires and how the PMT are placed. On the left side is illustrated a string with the doms.

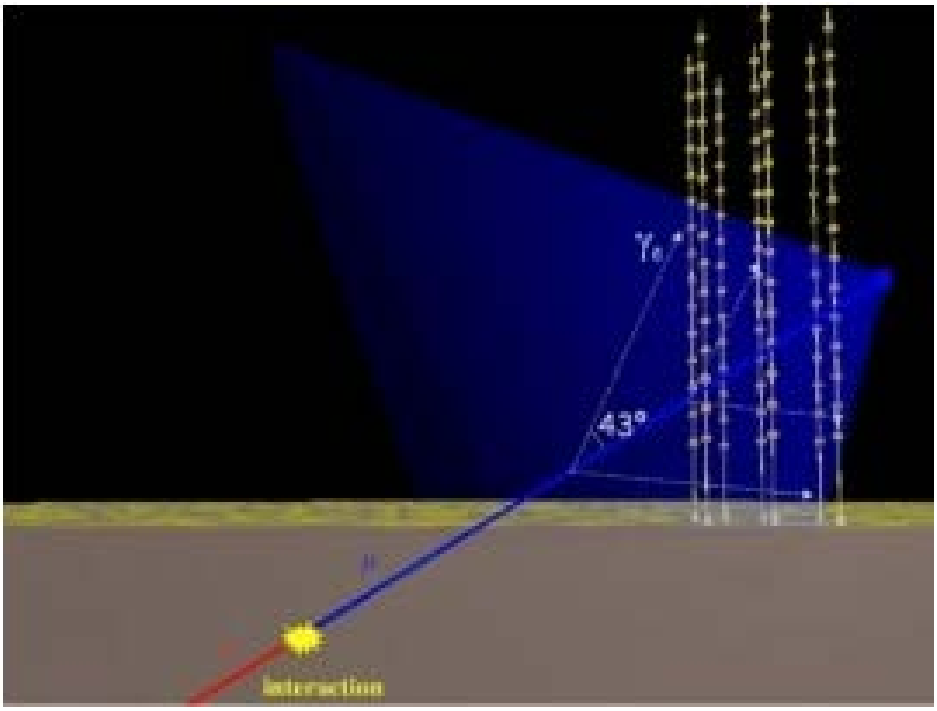


Fig. 1.3 The Principle of detection of high energy neutrinos in an underwater neutrino telescope (see text for details)

It used liquid scintillator and streamer tubes to detect charged particles and operated underground in the Gran Sasso laboratory in Italy. experiments are also being developed based on the detection of radio waves or sound produced in neutrino interactions. These techniques become viable for neutrinos with energies roughly above 10^{18} eV.

1.2 Detection Principles

The detection of high-energy muon neutrinos exploits:

- The emission of Cherenkov light by the muon and other charged secondary particles produced in a neutrino interaction;
- The directional correlation of the muon and parent neutrino trajectories to within 0.3° for $E_\nu > 10\ TeV$;

- The fact that upwardgoing muons can only originate from local neutrino interactions since the Earth filters out all other particles;
- The long range of muons in water and rock in the energy range of interest. As a result, muons may be generated far from the instrumented volume and still be detected.

1.2.1 Background

Backgrounds in a neutrino telescope are caused either by random light, not associated with particles traversing the detector, or by muons or neutrinos generated in cosmic ray interactions in the terrestrial atmosphere. The muons ("atmospheric muons") can penetrate the water above the detector and give rise to a reducible background. The neutrinos ("atmospheric neutrinos"), on the other hand, are an irreducible background.

Random Backgrounds

Daylight does not penetrate at any detectable level to depths beyond a kilometre. Sea water however contains small amounts of the naturally occurring radioactive potassium isotope, ^{40}K. This isotope decays mostly through β decay releasing electrons that produce Cherenkov light and produce a steady, isotropic background of photons with rates of the order of 350 Hz per square centimetre. Although the induced number of photo electrons per photomultiplier during the time it takes a muon to pass the detector (a few microseconds) is moderate there is still a chance that these hits may mimic the signature of a muon or shower or, more importantly, contaminate the hit pattern of a neutrino induced event. Many life forms that inhabit the deep sea emit light. This bioluminescence has two contributions, a continuous component usually attributed to bioluminescent bacteria,

and a component of localized bursts of light with high rates probably connected to macroscopic organisms passing the detector. These random backgrounds can be reduced by coincidence methods t to an acceptable level.

Atmospheric Muons

Cosmic rays entering the atmosphere produce extensive air showers which contain high energy muons. Although the sea water above the detector serves as a shield many such muons reach the detector. This background is reduced by deploying a neutrino telescope at great depth. The remaining flux of atmospheric muons is still many orders of magnitude larger than any neutrino induced muon flux. Since the atmospheric muons come from above this can be exploited to eliminate e them. However, multiple coincident atmospheric muons can produce a hit pattern that resembles that of an up-going muon. Therefore, care must be taken in the detector design and the reconstruction algorithms to minimize the rate of these fake events.

Atmospheric Neutrinos

Large numbers of charged pions and kaons are produced in cosmic ray interactions in the atmosphere. Their subsequent decays produce neutrinos, resulting in a large flux of atmospheric highenergy neutrinos. They are an irreducible background for the detection of neutrinos of cosmic origin. To study neutrinos arriving from cosmic point sources, a small search cone, commensurate with the angular resolution, is used. This reduces the background from atmospheric neutrinos to a manageable level. For the investigation of any diffuse neutrino flux of cosmic origin one relies on the fact that their expected energy spectrum is harder than the spectrum of atmospheric neutrinos. Thus it is possible to search

for an excess of cosmic origin neutrinos at higher energies. However, this may be complicated by the presence of the so called “prompt” atmospheric neutrino flux with a hard energy spectrum, arising from the decay of charm particles in atmospheric showers.

1.3 The Experiment challenges

The challenge of this project is the detection of high-energy neutrinos from the cosmos (see chapter 1). In order to achieve this challenge a deep study of the data flowing is necessary. In order to understand the reason of the decisions taken for developing the DAQ (Data Acquisition) for this experiment, is necessary to investigate how the detector produce data, which are the amount of data that is necessary to analyze, how is possible to achieve an online analysis of the data and how is possible to understand that data are containing valid information related to the experiment target.

1.3.1 Data Flowing Challenge

The offshore detector harvests the data starting from the OpticalModules (PMTs). Each PMT with its own logic board (FEM) is connected to a *Floor Control Module (FCM)* as shown in Fig. 1.4.

The FEM translates analog signal coming from the PMT into a digital signal. The logic board recognize the “excitement” of the PMT and communicates to the FCM when it happens and for how long. A “excitement” is called *hit*. A hit is a digital data composed as described:

- **PMT Info**: the ID of the PMT where the hit is occurred.
- **Hit time**: the time that the hit has lasted.

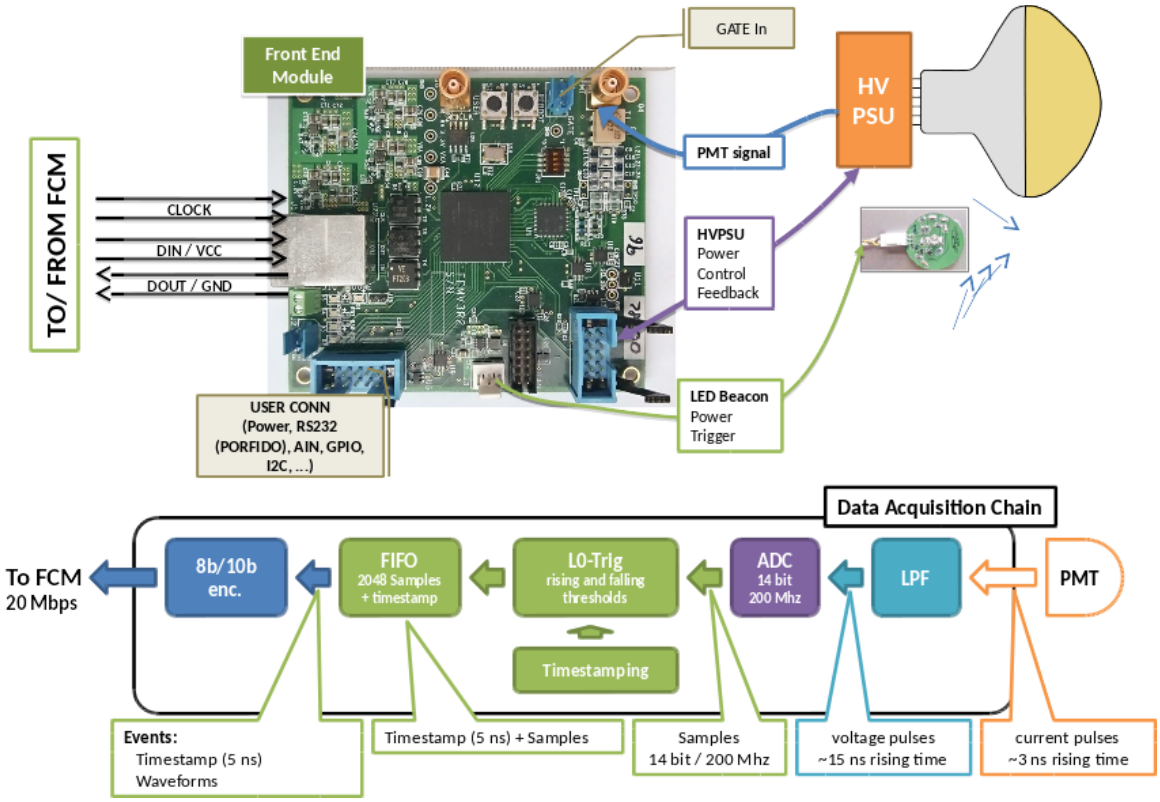


Fig. 1.4 Scheme of a PMT connection, the logic board shown is the FEM

- **Hit Charge**: a value that indicates how “bright” the hit has been.
- **Wave Form** A digital representation of the the sampled wave form generated by the hit.

The connection between the FEM and the FCM is serial with a custom protocol designed by the researchers of the FCM group. At this time the FCM receive these data from all PMTs of the floor (6 PMT per floor) continuously due the optical background and the wanted neutrinos reaction. [7]

The FCM is connected to a *junction box* that multiplex the signal of each FCM connected in order to use less optical fiber to bring the data to the shore station. In the shore station there is a respective demultiplexer that split the signals and send to the respective $NaNet^3$ logic board. Virtually each FCM is connected directly with the board with a two way communicating fiber. The fiber brings undersea the GPS signal, the clock, the Slow Control commands (e.g. “turn on”, “state”, “turn off”...) to the FCM along the path “shore to offshore” and on the other side it receives the stream of data. The protocol that FCM uses to communicate with the $NaNet^3$ logic board is called G-link. It is a synchronous protocol for data exchanging over fiber optics. A single $NaNet^3$ board can serve up to 4 FCM simultaneously. The FCM uses the GPS signal and the clock to mark the data with the timestamp. It encapsulates the data into structures called *DataFrame* before sending them to the offshore server. By design a DataFrame can have a variable dimension from a minimum of 8 to a maximum of 44 WORDs[1] (88 bytes).

On the shore machine there is a collaboration between two elements: the $NaNet^3$ and a software called *FCMServer*. The $NaNet^3$ write the data coming from the FMC on the RAM of the machine. The FCMServer handle two task, it opens a

[1] 2 bytes–long words

TCP socket and waits for a connection. When a client connects to the FCMServer software starts reading the data contained into the RAM and generates a data flow to the clients (see section 2.1). The *FCMServer* software generates standard

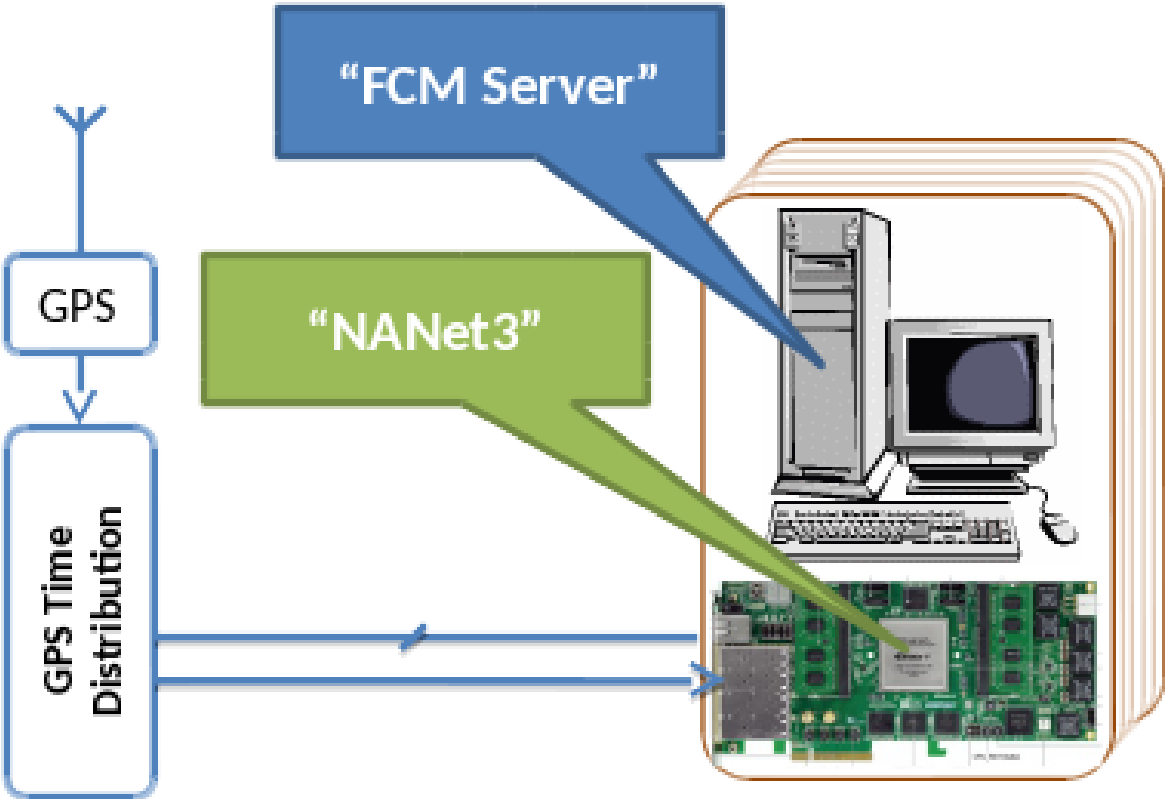


Fig. 1.5 Illustration of FCMServer

TCP/IP packets which payload is the *DataFrame*. At this point the acquisition can start and here is where the TriDAS acquisition system is connected to.

1.3.2 The Throughput Challenge

The detector beside the desired detection of the Cosmo's Neutrinos is affected by a background noise called *Optical Background Hit Rate*. It is a noise of false hits, they are generated by the environment due ^{40}K decay's. When a ^{40}K particle decays it generates a Muon that decays in a photon that is detected from the PMT. Fortunately is possible to calculate this kind of events. In the Mediterranean Sea

and at a depth of ~3500m the noise has a typical value of 50KHz with a maximum of 150KHz, in a conservative vision a value of 70KHz has been taken in account.

A DataFrame dimension is strictly related to the fraction of PMTs that are seeing photon in a certain moment. The worst case is when all the PMTs are seeing photons simultaneously for a long time. This means that the DataFrame will have the maximum size of 88 bytes (as introduced in section 1.3.1), due to the sum of background noise and the actual neutrinos signals, but the frequency of the neutrino data is very poor, covered by the noise.

If this happen the detector will have a throughput calculated as follow:

DataFrame dimension $\rightarrow$ DFD

Optical Background $\rightarrow$ OB

$$DFD * OB * N_{PMT\ FLOOR} * N_{FLOOR\ TOWER} * N_{TOWERS} = throughput\ in\ bps$$

Considering the worst case, a PMT generates 88 bytes of data.

Every PMT is affected by the noise on 70KHz ^{40}K decay .

$$88\ bytes\ *\ 70KHz\ =\ 6.160.000\ Bi/s \rightarrow 49.280.000\ bps \rightarrow 49,28\ Mbps$$

On a floor there are 6 PMTs.

$$49,28 Mbps *\ 6\ =\ 295,68\ Mbps$$

In a Tower there are 14 floors.

$$295,68\ Mbps *\ 14\ =\ 4139,52\ Mbps \rightarrow 4.14\ Gbps$$

The entire detector will be composed by 8 Towers.

$$4.14\ Gbps * 8 =\ 33,12\ Gbps \rightarrow 3.86\ GiB/s$$

In this case it has been considered that all the PMTs are registering a hit at the same time. This is quite impossible due the Poisson distribution of the hits. A quite real averaged value of a DataFrame dimension is 46 Bytes.

Repeating the various calculation with this new value a more realistic throughput is shown in Table 1.1.

Table 1.1 Expected throughput from the Detector

Case (KM3NeT-Italia)		**Expected** ($v_{single} = 50kHz$)	**Conservative** ($v_{single} = 70kHz$)	**Maximum** ($v_{single} = 150kHz$)
10” PMT (0.25 p.e thresh)	(Mbps)	19.0	26.0	56.0
floor (6 PMT / floor)	(Mbps)	110.0	160.0	330.0
Std Tower (14 floors)	(Mbps)	1600.0	2200.0	4700.0
NEMO Phase 2 (8 floors - 4 PMT/floor)	(Gbps)	0.6	0.8	1.8
Full Detector (8 Std Tower)	(Gbps)	12.0	17.0	37.0

1.3.3 The Computational Challenge

As described, during an acquisition phase a lot of data arrive at the offshore station. It is easily to imagine that computing all the data, that the detector is producing, is a hard task. Although, for computing the data and detect the significant neutrinos traces the trigger algorithms need to be aware of the whole detector. This mean that, ideally, we need to aggregate all the data arriving from each PMT in a unique stream, sending it to a single computer for the processing. Obviously, this is impossible to achieve without some intermediate steps and some kind of distribution of the workload over multiple nodes. In first instance, it is necessary to have a complete sight of the entire active detector, this forces to manipulate the real-time stream and splitting it in “pieces” or more precisely “slices”. The idea is to slice the stream in packets that contain a certain amount of data related to a customizable time-window. This approach permits

to process “snapshots” of the acquired data from the detector and after a first aggregation level the distribution of this aggregated data through several nodes, this snapshot is called TimeSlices. Furthermore, this kind of approach allows to scale and adapting the TriDAS to a detector that could increase its dimension along years, resolving in advance scalability problems like the large amount of data arriving at the offshore station. Thus, when the detector will increase the number of the towers the system will not hit the physical communication limits. With the *Time-slicing* method could happen to splitting a relevant “event” (an “hit” sampled from the PMTs) in two different TimeSlice. In this case is necessary to define how large must be the slice in time. During the studying and simulation of the detector this dimension has been calculated and this is 200ms. Indeed, with this value the probability to split a relevant event in two TimeSlice is 10^{-6}. This probability is small enough to be acceptable for the experiment. Moreover, with this slice dimension the TriDAS can handle the online triggering with the current technology limitations. The experiment takes into account the costs too, in fact there are technologies that could permit to handle different dimension of TimeSlices and throughput but they are too expensive and not sustainable on the long term.

1.3.4 The analysis Challenge

It has just been said that the data is being “sliced” because is necessary to have a complete sight of the detector for discriminating the relevant neutrinos trace from the background. For achieving this task is necessary to use two different algorithms. An L1 level and a L2 level algorithms. These algorithms are called *Trigger*. The L1 algorithms are hardcoded into the TCPU (see section 2.3) program. A L1 trigger consist of a logic OR of the following conditions:

- Simple Coincidences: i.e. topological time-like trigger conditions involving hits occurring on near PMTs within a time-window of 20 ns. The circles in Fig 1.6 indicate the pairs of PMTs of each floor that can trigger Simple Coincidences L1 events.

- Charge over threshold: when a hit charge is above a certain threshold in picoCoulomb (note that 1 single photon electron hit charge is about 8 pC).

- Floor Coincidences: when one PMT has a coincidence with another PMT that is not in the same couple but it is on the same floor.

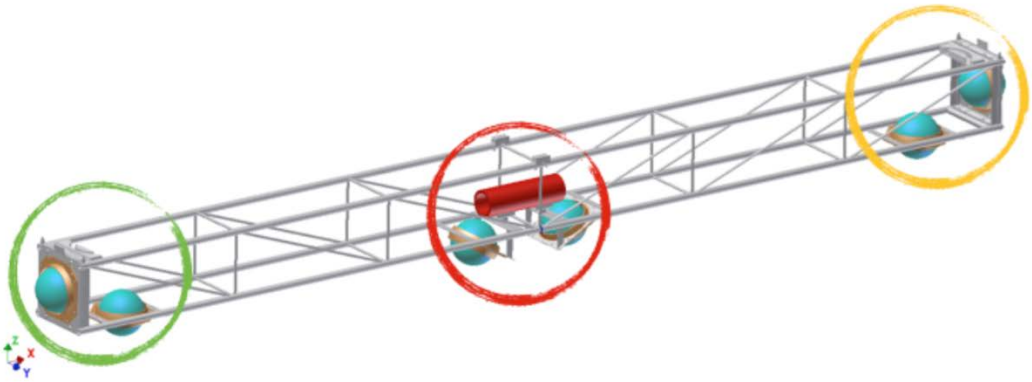


Fig. 1.6 The circles groups PMTs in pairs: each pair could trigger a Simple Coincidence.

When a trigger seed has been found in the stream of one TimeSlice, a L1 event is cut, saving all the hits from all the PMTs within $\pm 3\mu s$ from the trigger seed occurrence. If a new seed occurs before the end of the event, the event time-window is extended by counting additional $3\mu s$ from the last found seed.

The Level 2 trigger consist on a causality filter [8]. Only L1 events are containing a number of L1 seeds $\geqslant 5$ were considered. The strategy consist of testing the causality among the hits of the L1 seeds assuming being originated

by a muon coming from a guess direction out of a collection of 210 ones which cover the full solid angle 4π (see Fig. 1.7).

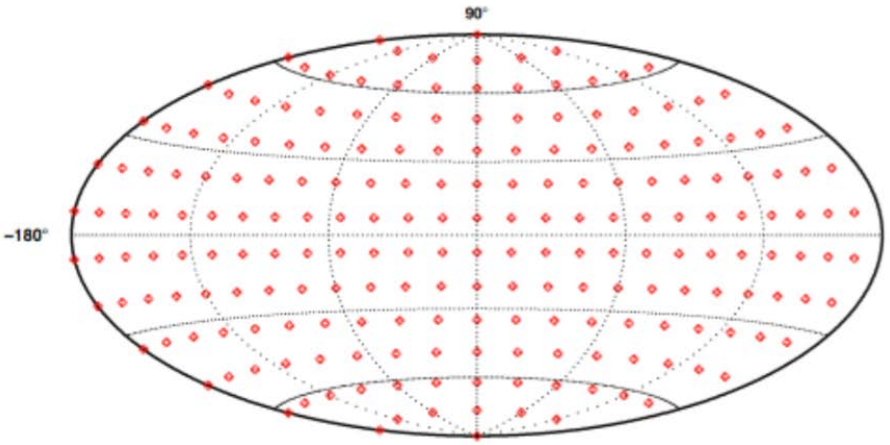


Fig. 1.7 Hammer-Aitoff projection of the default grid of 210 directions used in the standard trigger algorithm.

The L2 algorithm proceeds as the following: given the direction, the detector frame is rotated till the chosen direction becomes the vertical one. Then the causality relation was tested with equation 1.1 for all the pairs of L1 hits (refer also to Fig. 1.8):

$$|(ti - tj)c - (zi - zj)| \leqslant tan\theta_c \sqrt{[(x_i - x_j)^2 + (y_i - y_j)^2]} = tan\theta_c |R_{ij}| \qquad (1.1)$$

The application of the L1 + L2 trigger levels determines a trigger efficiency ranging 3-5%. It is a reasonable efficiency considering that the triggered good events are, for this simulation, downward going atmospheric muons, and being the NEMO-like towers optimized for the detection of upward going particles. Moreover, approximatively the same trigger efficiency is found for the ANTARES detector [9], which has almost the same instrumented volume.

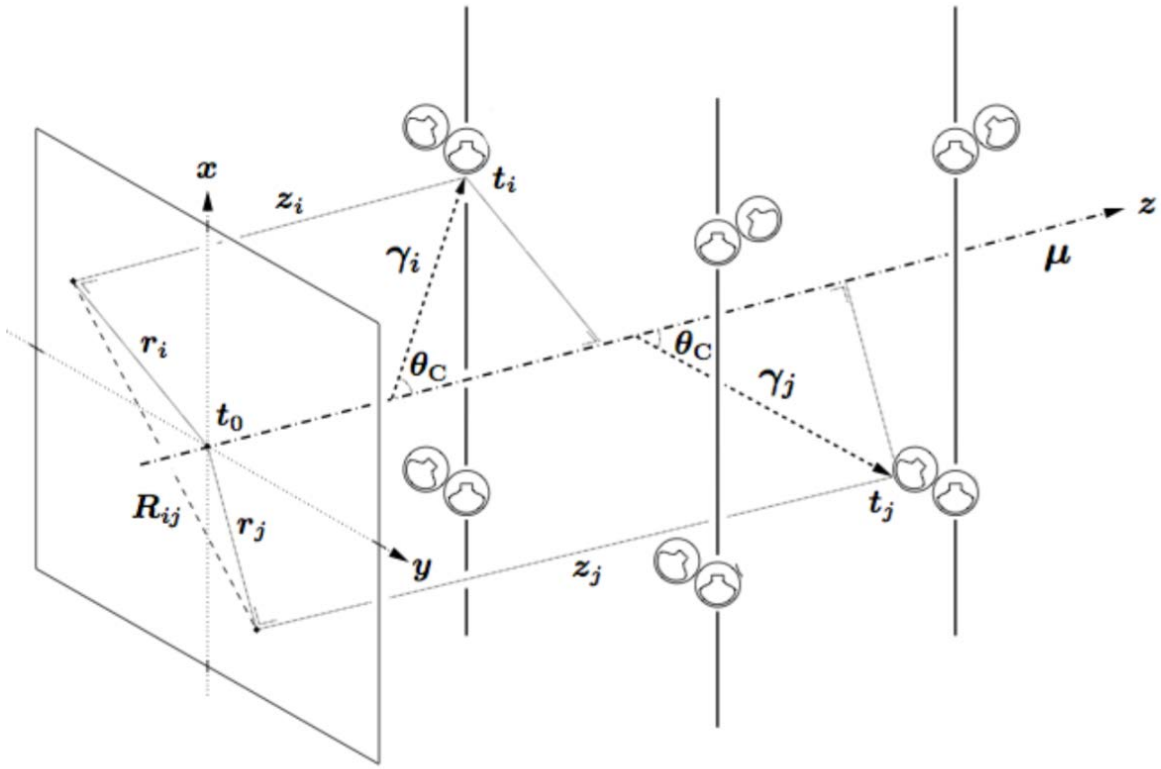


Fig. 1.8 Schematic view of a muon traversing a part of the instrumented volume of the detector.

Chapter 2

Design and implementation of the Trigger and Data Acquisition System

The TriDAS Core [10] (Figure 2.1) is formed by the HitManager (HM), the Trigger CPU (TCPU), the TriDAS-SuperVisor (TSV) and the Event Manager (EM).

In figure 2.1 and 2.2 there is a schematic representation of the TriDAS.

The whole system derives from a base developed for the previous phase of the experiment, in this version the modularity of the system has been kept but most part of the system has been redesigned in order to be more scalable and reliable for the foreseen multiple towers.

The FCMServer (FCMS) units provide data to the TriDAS core, these servers read the PMT data from the detector and send them to the HitManager. The FCMServers are designed to be the onshore gate for all the kind of data streams (slow control, optical and acoustic), going to and coming from the offshore detector [11]. One single FCMServer can handle the optical data coming from 4 floors of a Tower. With 8 Towers, the total number of FCMServers is 32. The

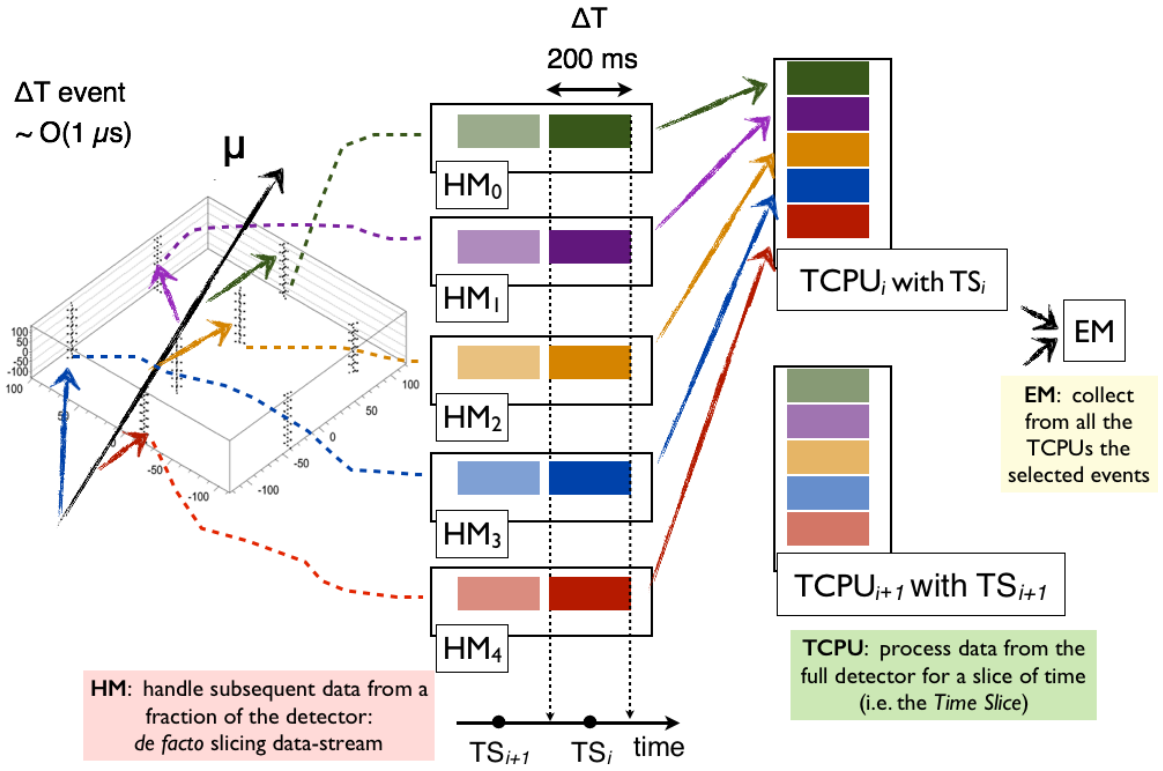


Fig. 2.1 The TriDAS core overview

FCMServers forward the data coming from the OMs to the first layer of the TriDAS, the HitManagers.

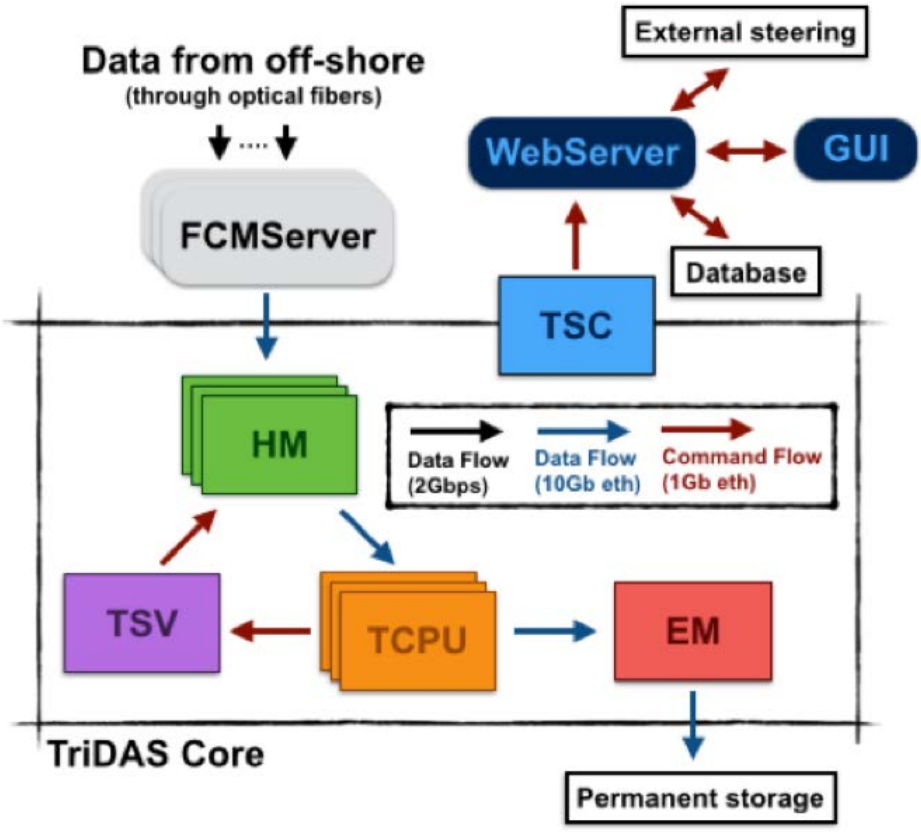


Fig. 2.2 The TriDAS core overview

Every single HitManager process runs in a dedicated server and it is linked to a fixed number of FCMServers ,which correspond to a portion of the detector, called Sector. All the HitManagers share the same time line, originated from a common timestamp, which is quantized in subsequent intervals of time of the same duration, called *TimeSlices*. In this way, a full set of PMT data occurred during a particular TimeSlice are asynchronously managed by the HitManagers that is managing that FCMServer, the HitManagers organize their own fraction of data in a special data-structures called the "*Sector Time Slices*" (STSs). The

role of the TriDAS-SuperVisor is to steer all the HitManagers sending the STSs belonging to the same TimeSlice to the first available TriggerCPU, according to a free-token-scheduler mechanism. In its turn, one TCPU collects all the STSs of a TimeSlice into the so called "*Telescope Time Slice*" structure (TTS), then process it according to the trigger algorithms [4]. Many TriggerCPUs process different TTS at the same time. The fraction of data which fulfill the trigger selection criteria is sent to the EventManager, which records the filtered data in binary files on the local storage. Offline, the written post-trigger files are transferred from the Shore station infrastructure of Portopalo to the storage facility at LNS via a dedicated 10 Gbps connection. The design for TriDAS is modular and scalable with the number of deployed Towers. The required amount of TriggerCPUs processes depends on the complexity of the trigger algorithms and increases with the number of OMs.

2.1 FCM Server

The *Floor Control Module Server* called *FCMServer* is a program that works in collaboration with a $NaNet^3$ card [11], as introduced in section 1.5.

The $NaNet^3$ is a logic board plugged on the PCIe[1] bus of a server that reads the optical data from an undersea tower's floors and after a decoding it writes that data on the RAM of the pc. A $NaNet^3$ can read and manage the data coming from at most 4 floors. This part of software is developed by a collaborating group in Rome. From the point of view of the FCM Server program, it spawns a different network socket for each different floor. Therefore, we treat every

[1] PCI Express® (Peripheral Component Interconnect Express), officially abbreviated as PCIe®, is a computer expansion card standard designed to replace the older PCI, PCI-X, and AGP standards. It is used to link motherboard-mounted peripherals and as an expansion card interface for add-in boards.

socket as a “different” FCM server when we are talking about “FCM Server” from the *HM* side. When a client connects to a socket the FCM server starts to send a continuous stream of data to its 2.3. This data are blocks that contains this information:

- GPS timestamp
- Data from each OpticalModule
- other data

The GPS timestamp is the key information that will be used on the next block chain.

As told previously the next element that read the data from the FCM server is the HM.

A FCM serves only a HM for the whole configuration.

Fig. 2.3 A FCM Board

2.2 Hit Manager

The purpose of the Hit Manager is to transform/slice the data in blocks according to an absolute time-"label". The label that has been used is the absolute time contained in each segments of the data received from the FCM Server (2.1). In fact, the data that the FCM is sending is like a stream, but that stream is structured and the data from the floors are all synchronized with a global timestamp calculated from a GPS with the nanosecond precision. The HMs

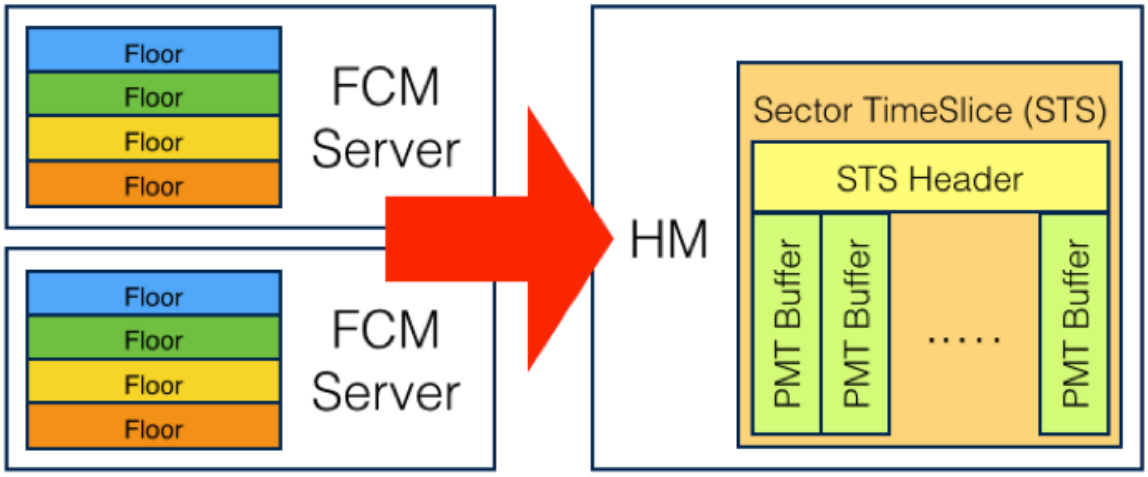


Fig. 2.4 FCM to HM data transfer

represent the first aggregation stage for the incoming data-stream. Each HM handles a number of floors (Fig.2.4) or FCMServer and all these represent a "Sector", slicing data in subsequent TimeSlices (TS) of the same fixed duration and referred to a common time origin. Each HM organizes its own sliced data in special structures called SectorTimeSlices (STSs) and sends them to the TriggerCPUs. At every "Run" a number of HM are running according to the number of floors that the FCM Server are serving.

More floors are present during a Run more HM could be necessary. Each HM maintain a customizable buffer of STSs inside itself for multiple reasons:

1. The HM is not synchronized with the second step of the TriDAS analysis (TCPU2.3), in fact an analysis of the detector could be slower or faster depending on what the Telescope has detected in a certain moment.

2. it could arrive an external trigger

Accordingly with the design of the international collaboration of acquiring data from the universe, the Detector can receive an external trigger. This trigger is emitted from the international collaboration when a transient astrophysical phenomena such as Gamma Ray Bursts (GRB) or SuperNova explosions are detected. In this case is required to save at least 30 minutes of recorded data from the detector to permanent storage. When this request arrives to the TriDAS the HM is informed to store all the data that it is keeping, sending them directly to disk. In order to be always ready to satisfy this request every HM keep a time window 30 minutes wide in an internal buffer. The number of how many STSs the HM must keep for covering the 30 minutes of data is defined into the Datacard in which is defined how big is a “slice” of the data. The Datacard is a single configuration file that is shared from all the components of the TriDAS at the beginning of the acquisition. In this file there are sections related with each component of the TriDAS and the description of the geometry of the detector.

Listing 2.1 Datacard Part related to ALL components configuration

```
1  "INTERNAL_SW_PARAMETERS": {
2    "DELTA_TS": "200",
3    "PMT_BUFFER_SIZE": "1000000",
4    "STS_READY_TIMEOUT": "5",
5    "TTS_READY_TIMEOUT": "30",
6    "STS_IN_MEMORY": "100"
7  },
```

As is possible to see into the listing 2.1 there is a “key” value named *INTERNAL_SW_PARAMETERS*. Inside this block there are many configuration and HM related are:

- **DELTA_TS**: this number is in milliseconds format, it tells to the HM how big is the slice of the data that it must create, typically, 200ms.

- **PMT_BUFFER_SIZE**: this number is expressed in bytes. It indicates the maximum buffer size deputed to contain the data of a PMT in a TS.

- **STS_READY_TIMEOUT**: this number is correspond to seconds. It is a timeout that indicates how much the HM must wait before consider a STS as ready to be sent.

- **STS_IN_MEMORY**: this number is an absolute number that tell to the HM how big must be the internal buffer of STS for the reason explained before (GRB).

Listing 2.2 Datacard Part related to HM configuration

```
"HM": {
  "LOG_LEVEL": "DEBUG",
  "BASE_CTRL_PORT": "16100",
  "DUMP_FLAG": "0",
  "DUMP_FILENAME_PREFIX": "\/tmp\/hm_dump_",
  "DUMP_MAX_SIZE": "500",
  "HOSTS": [
    {
      "CTRL_HOST": "192.168.253.114",
      "N_INSTANCES": "1"
    },
```

In the second listing 2.2 there is listed the configuration of the for the HM.

- **LOG_LEVEL**: indicated the verbosity of the log that the HM will produce during the run (possible options: DEBUG | INFO | WARNING).
- **BASE_CTRL_PORT**: is the BASE port number where the HM wait the connection from the TSV (2.4) in order to receive the message. A message contains this information:“the STS containing this TS must be sent to this TCPU”. The actual port will be calculated on launch, see Section **??**.
- **DUMP_FLAG**: these are flags for debugging purpose, when this is set to “1” the HM will dump all the received data to a file with a prefix defined on *DUMP_FILENAME_PREFIX* key.
- **DUMP_FILENAME_PREFIX**: is the prefix of the file when the HM must write on disk the dumped data.

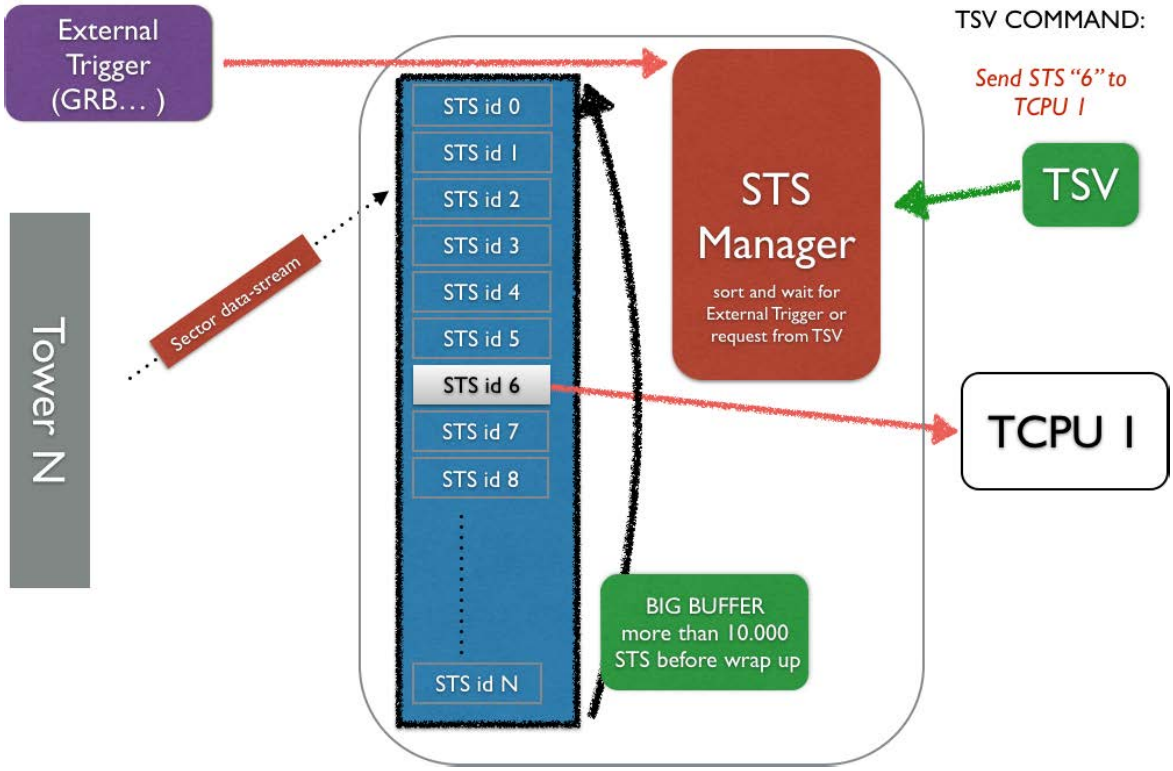


Fig. 2.5 HM block Scheme

- **DUMP_MAX_SIZE**: debug value that indicates how many STSs must be dumped.

- **HOSTS**: inside this key are indicated all the instances of HM that must be launched in this "Run" over all the HM hosts.

  - **CTRL_HOST**: is the IP where the HM will wait the connection from the TSV.

  - **N_INSTANCES**: is number of instances of HM (different process) that host must have running during the "run".

As is shown in Fig.2.5 the HM has is connected with other two TriDAS component: TCPU (2.3) and TSV (2.4). Following the TriDAS chain, as a matter of fact, we must to deliver these data to some one that has the ability to reconstruct

the entire detector and understand if the data is meaningful. But the HM contains only a part of the data of the entire detector. Is necessary, indeed, that the TSV tell to the HM where to send which data. In other words, the HM waits for a message from the TSV where is indicated which STS must be sent to a single TCPU instance. When this message is received the HM check if the required STS is present on the buffer and then send the STS directly to the TCPU. The STS is not discarded yet, for the external trigger reason, but it will discarded when the buffer will wrap up.

In order to not overload the TCPU workload the HM is always waiting for a message from the TSV before send any data to the TCPU. The HM wait for the connection from the TSV in order to let to him the control of the data flowing. If the TSV is not running the HM will retry indefinitely to reconnect to the TSV. This permit to be ready to start an acquisition (see Section 3.1).

2.3 Trigger CPU

The Trigger CPU better known as TCPU is responsible for the last step of data aggregation and online analysis. A TCPU receives the STSs from HMs creating a TotalTimeSlice (TTS),then it applies triggers to this new object and finally sends it to the Event Manager. As we shown in fig.2.6, is possible to identify six elements with which the TCPU is composed.

In first instance, the TCPU receives the STSs from several HMs and via the “TTS Builder” the program aggregates them. The “TTS Builder” has the task of reassemble all the STSs belonging to the same TimeSlice coming from all HMs into a single “TotalTimeSlice” or “TTS” data structure. When the rebuild process is complete the TTS is containing the data from the whole detector with a wide of the slice width (e.g. 200ms). This block allocates memory dynamically trying

to reuse as much memory as possible. It uses a queue where take already freed memory called “TTS Free”, if this queue is empty it will allocate new memory and a new TTS will be created. When the data structure is loaded with the incoming data it will be marked with a TTSID that is the ID of the TS[2].

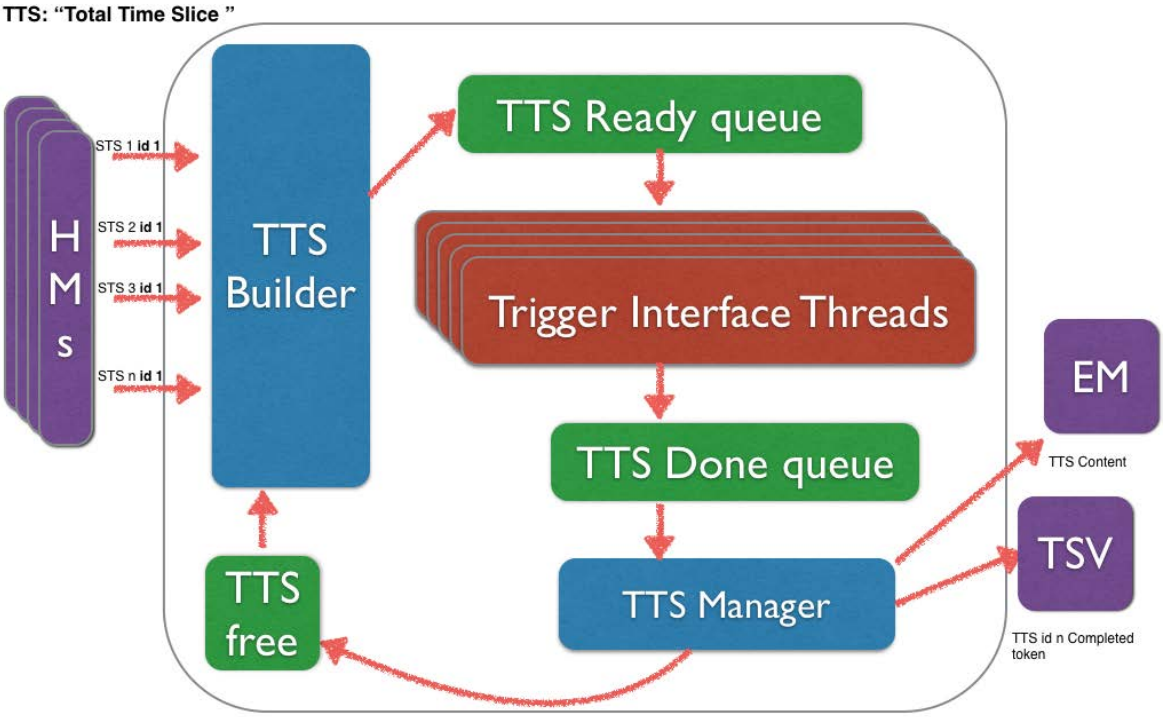


Fig. 2.6 TCPU Block scheme

At any time, the TCPU can have only a certain amount of allocated memory due the fact that there are only a limited number of “token”.

It could happen that a HM is receiving the data with delay or, even worse, is not receiving the data at all from the FCM connected server, so when a STS is requested and it must send the required STS to a TCPU, it is possible that the HM will not reply to the request. The TCPU can manage this kind of situation by using a variable inside the *INTERNAL_SOFTWARE_PARAMETER* block, as shown into the listing 2.3:

[2]TimeSlice

- **TTS_READ_TIMEOUT**: this value is expressed in seconds and indicates how many seconds a TTS must be kept into the “TTS Builder” before marking it complete and moving it to the analysis block.

Listing 2.3 Datacard Part related to ALL components configuration

```
"INTERNAL_SW_PARAMETERS": {
  "DELTA_TS": "200",
  "PMT_BUFFER_SIZE": "1000000",
  "STS_READY_TIMEOUT": "5",
  "TTS_READY_TIMEOUT": "30",
  "STS_IN_MEMORY": "100"
},
```

This timeout starts when the first STS related to a TS (TimeSlice), is received. In both cases, either the TTS has been completed or the timeout has expired, the TTS is moved into the “TTS Ready Queue” for further data processing. If a STS that should have belonged to an already moved TTS is received, it will simply be discarded.

As the name of this part of the TriDAS suggests, the main task of the program is analyze the TTSs and try to find out if the data are relevant or not.

The TCPU as the HM uses a part of the Datacard, as is shown in the listing 2.4.

Listing 2.4 Datacard Part related to TCPU component configuration

```
"TCPU": {
  "LOG_LEVEL": "DEBUG",
  "DUMP_FLAG": "0",
  "DUMP_FILENAME_PREFIX": "\/tmp\/tcpu_dump_",
  "DUMP_MAX_SIZE": "500",
```

```
    "BASE_CTRL_PORT": "16200",
    "BASE_DATA_PORT": "16300",
    "OFFLINE_FLAG": "0",
    "SIMULATION_FILENAME": "\/tmp\/simulated_events.txt",
    "PARALLEL_TTS": "2",
    "PLUGINS_DIR": "\/tmp\/plugins",
    "HOSTS": [
      {
        "CTRL_HOST": "192.168.253.118",
        "DATA_HOST": "10.0.80.118",
        "N_INSTANCES": "1"
      },
      [CUT]
    ],
       "TRIGGER_PARAMETERS": {
            "L1_EVENT_WINDOW_HALF_SIZE": "600",
            "L1_DELTA_TIME_SC": "4",
            "L1_DELTA_TIME_FC": "20",
            "L1_CHARGE_THRESHOLD": "500",
            "L1_FLAG_RT": "1",
            "L1_FREQUENCY_RT": "5",
            "L1_DELTA_TIME_RT": "200000",
            "L1_DELTA_TIME_SEQHIT": "200",
            "L1_N_SEQHIT": "7"
        },
        "PLUGINS": {
```

```
32              "RANDOM": {
33                  "NAME": "TrigRandom",
34                  "ID": "0",
35                  "PARAMETERS": {}
36              },
37              "SCALER_10": {
38                  "NAME": "TrigScaler",
39                  "ID": "1",
40                  "PARAMETERS": {
41                      "SCALE_FACTOR": "10"
42                  }
43              }
44          },
```

The meaning of each “configuration key” are defined as follow:

- **LOG_LEVEL**: indicates the verbosity of the log that the HM will produce during the run (possible options: DEBUG | INFO | WARNING).
- **DUMP_FLAG**: these are flags for debugging purpose, when this is set to “1” the HM will dump all the received data to a file with a prefix defined in the *DUMP_FILENAME_PREFIX* key.
- **DUMP_FILENAME_PREFIX**: is the prefix of the file when the HM must write on disk the dumped data.
- **DUMP_MAX_SIZE**: another debug value that indicates how many TTS must be dumped.

- **BASE_CTRL_PORT**: is the port where the TCPU waits the connection from the TSV (2.4) in order to send a message (token). A message contains this information:“I have finished the TTS related to this TS, I’m ready for receiving new data”.

- **BASE_DATA_PORT**: is the BASE port where the TCPU waits the connection from the HMs for receiving the STSs. It is the start number for calculating the right port where to wait the connection.

- **OFFLINE_FLAG**: this is a boolean value that indicates to the TCPU if it must run in “online” mode or “offline” mode (see Section 2.3.1).

- **SIMULATION_FILENAME**: is the path of the file to use during an offline analysis.

- **PARALLEL_TTS**: is the number of parallel TTS (how many threads) the TCPU must manage during its run.

- **PLUGINS_DIR**: is the directory where the TCPU will search the plugin to dynamically load during at launch.

- **HOSTS**: this “key” contains an array with the following keys:

  – **CTRL_HOST**: is the IP where the TCPU must wait for the connection from the TSV.

  – **DATA_HOST**: is the IP where the TCPU must wait for the connection from the HMs.

  – **N_INSTANCES**: is the number of instances of TCPU (different processes) that the host must have running during the “Acquisition Run”.

- **TRIGGER_PARAMETERS**: the TCPU has two levels of trigger, L1 and L2. In the list below, the keys related to the L1 trigger will be described in details. This trigger is hardcoded into the TCPU itself:
  - **L1_EVENT_WINDOW_HALF_SIZE**: this value indicates the amount of time-data to keep before and after a Hit while building an event[3].
  - **L1_DELTA_TIME_SC**: this value will determine how much time can be passed between two hits coming from two adjacent PMT. When this happen a *simple coincidence* Trigger will be activated. This value has the format of μs.
  - **L1_DELTA_TIME_FC**: maximum time between two Hits coming from two PMT that do not belong to the same couple of adjacent PMT; if the time is under this threshold a *Floor Coincidence* Trigger will be activated. This value is represented in μs.
  - **L1_CHARGE_THRESHOLD**: threshold of minimum charge detected by a PMT[4] in order to consider the hits as relevant. This value is represented as a number.
  - **L1_FLAG_RT**: this is a flag activates a particular trigger algorithm that randomly identifies a bunch of data as an Event. This particular behavior is needed in order to refine during the offline analysis the trigger algorithm, this data is used as *control sample*. This value is a boolean.
  - **L1_FREQUENCY_RT**: with this key it is possible to indicate how often the above *Random Trigger* will be activated. The value is in Hz.
  - **L1_DELTA_TIME_RT**: it is the duration of a *Random Event* in μs.

[3] An Event is a bulk of hits that has activate the trigger algorthm
[4] Value present into the data

  - **L1_DELTA_TIME_SEQHIT**: this key is for calibrating the “shower hits”. The value indicates a timewindow where a certain amount of hits must be present in order to activate the *Sequential Hit* Trigger.
  - **L1_N_SEQHIT**: the minimum amount of Hits that must be found on the above timewindow in order to activate the *Sequential Trigger*.

- **PLUGINS**: as told before the TCPU has two level of Trigger the L2 Trigger are defined into this “node”. The TriDAS has the capability to load multiple different Trigger plugin dynamically at launch time and their description is located here.

The structure of a plugin “node” is described below.

- **“NAME”**: this is a special key with human-readable form of the plug-in.
- **NAME**: name of the plugin that must be searched into the plugin directory and once found loaded
- **ID**: is a sequential order that indicates to the TCPU the absolute order of the launch of the trigger over each TTS.
- **PARAMETERS**: optional parameters for this plugin.

The TCPU is designed to analyze several TTS at a time so it is possible to scale on machines with multiple processors and cores. At launch the TCPU will prepare a bunch of threads that will concur on a queue. These threads will hang on the “get data” function of the *TTS Ready Queue*. This threads are represented by the *Trigger interface Threads* block on the diagram in Fig.2.6.

After that the TTS has been processed from the *Trigger Interface Block*, that will run the L1+L2 Triggers algorithms (see Chapter1.3.4). The analyzed TTS is

put in another queue called *TTS Done Queue*. Here the TTSs are processed from the last thread of the program: the “TTS Manager”. The purposes of this thread are three:

1. send the processed TTS from the trigger to the EM (see Section 2.5) for consolidating the results.

2. after data transmission, it cleans the data structure of the TTS and puts this “pointer” into a queue called *TTS Free* in order to reuse it in the *TTS Builder* with new data coming from the HM.

3. send a communication to the TSV that a TTS with a specific TSID[5] has been completed and that the TCPU has space for further data processing.

It is clear that the computational time of a TCPU is not constant since a TTS could contain more Hits than another and this means that their processing is more complex. A more complexity during the processing brings unavoidably to longer computation time. All the system is realized to process data in real-time during acquisition so is necessary to have some kind of “autobalacing” mechanism in order to let an overloaded TCPU the time to “digest” the data. The system implemented in this study is the token based mechanism. Each TCPU has a certain amount of tokens available and each token correspond to one and only one TTS that can be processed by each TCPU. The managing of the token is handled by the *TTS Manager* thread. At launch time it tries to connect to the TSV sending all the tokens that are defined. If the TSV is not running and the token exchange can not be done, this thread retries indefinitely until the TSV accepts the token. This permit the system to be ready to start the acquisition, this behavior is strictly coupled with the State Machine of the TSC (see section 3.1).

[5] TimeSliceID

During the acquisition phase every time a TTS is sent to the EM and the data structure cleaned, the *TTS Manager* sends a token to the TSV indicating which TTSID has been just finished. This mechanism brings the system to autobalance the load on the TCPU indeed if a TCPU is overloaded it will not send tokens, so it will not receive new TTS to process.

2.3.1 TCPU Offline

The simulation of the trigger is an important stage of the characterization of the trigger itself. It allows to determine the efficiency and purity of the trigger algorithms, providing also a basic — but powerful — testbed for their software implementation. The basic idea is to execute a real TCPU process feeding it with data contained in a file. The output of the trigger is written as a normal post-trigger file, as in the online context. The input and output file formats coincide. This has several implications: in first place, it allows to hide the nature of the data fed into the simulation process, unbiasing its behavior; secondly, it allows to handle a single data format, simplifying the code and its maintenance; moreover, it allows to re-trigger the data iteratively without any further effort, thus online-taken data can also be reprocessed.

The Trigger Simulator (*TrigSim*) is the software component deputed to the trigger simulation. The program recreates a minimal TriDAS environment to support the execution of a real TCPU process. So, the TCPU is the very same as the online one while the remaining TriDAS components (HM, TSV, TSC and EM) are simulated.

The *evt2pt* program translates a file from the ANTARES EVT format [12, 13], widely used in simulation chains by the ANTARES and KM3NeT Collaborations,

into a post-trigger file format. The Datacard field contains the simulation parameters.

2.4 TriDAS Super Visor

The TSV supervises the data exchange between HM and TCPU, taking note of the processed TSs. The TSV is one of the most simpler processes present into the TriDAS but it is one of the most important, indeed it informs all HMs to send their STS to a specific TCPU that will build the TTS for the data processing. In order to orchestrate the data-flow the TSV needs a “Start TS ID” that is a “Start Time”. This information is provided by the TSC at TSV launch time. Therefore, when the TSV starts it will connect to the HMs and TCPUs and wait the incoming data from the TCPUs. When a TCPU gets connected, it will send all its tokens. The TSV will receive all the tokens and for each token it will assign a new TSID starting from the received *Start Time*.

When a TCPU is ready to handle new data, it sends a token to the TSV. The TSV selects a TS ID among those not yet processed and communicates to all HMs to send the corresponding STS to that TCPU. The TSV keeps track of every TSID that is under processing and every time one TSID is completed it calculates the next ID that must be processed from every single TCPU.

Listing 2.5 Datacard Part related to TSV component configuration

```
1  "TSV": {
2    "LOG_LEVEL": "DEBUG",
3    "CTRL_HOST": "192.168.253.113"
4  },
```

The meaning of each “configuration key” are defined as follow:

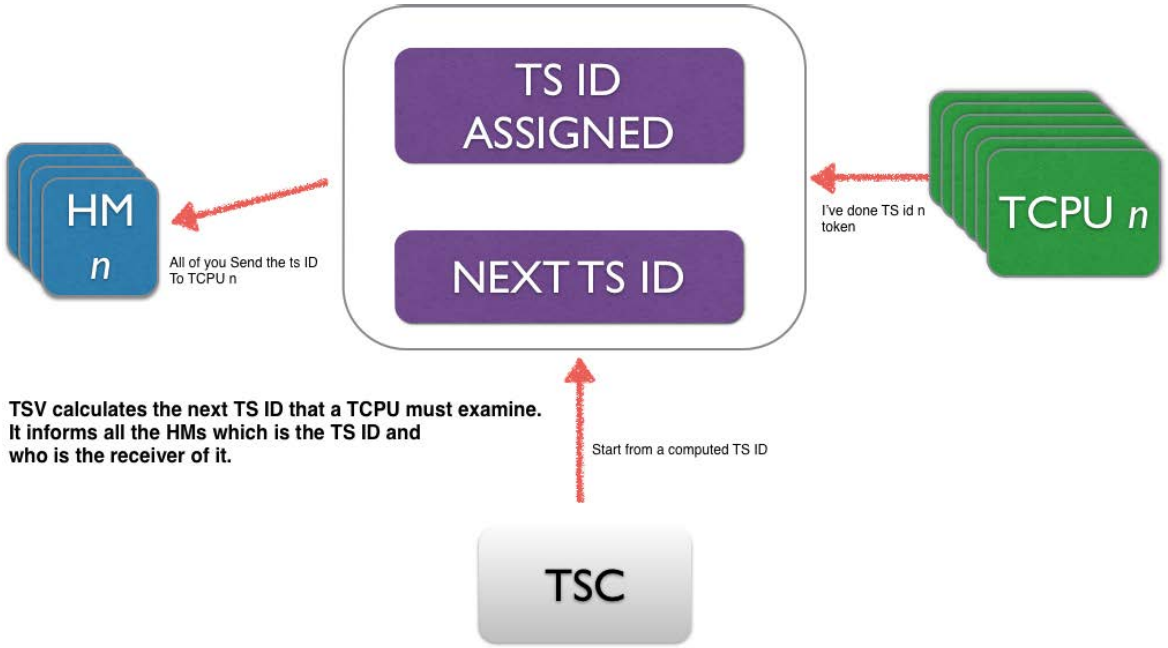


Fig. 2.7 TSV Block scheme

- **LOG_LEVEL**: indicates the verbosity of the log that the HM will produce during the run (possible options: DEBUG | INFO | WARNING).
- **DUMP_FLAG**: these are flags for debugging purpose, when this is set to "1" the HM will dump all the received data to a file with a prefix defined on *DUMP_FILENAME_PREFIX* key.
- **CTRL_HOST**: Indicates the IP of the machine that will run the TSV on the Control Network

2.5 EventManager

The EM is the software component of TriDAS devoted to the storage of triggered data. A single EM process collects triggered data from the whole TCPU set and performs data writing on local storage.

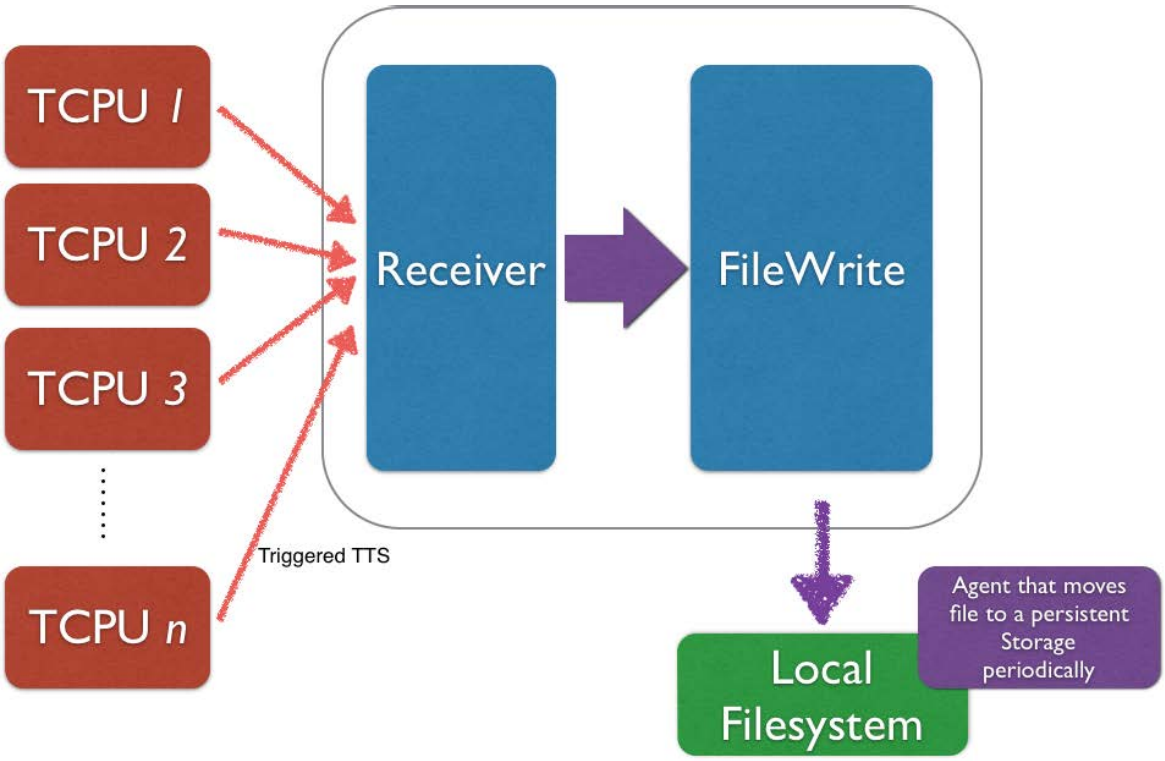


Fig. 2.8 EM Block Scheme

Listing 2.6 Datacard Part related to ALL components configuration

```
"EM": {
  "CTRL_HOST": "192.168.253.112",
  "DATA_HOST": "192.168.253.112",
  "DATA_PORT": "16400",
  "NETWORK_THREADS": "1",
  "LOG_LEVEL": "DEBUG",
  "LOG_TO_SYSLOG": "0",
  "FILE_MAX_SIZE": "2000000000",
  "PT_FILE_PREFIX": "/home/tridas/nemo_f3_pt",
  "PT_FILE_POSTFIX": ".dat"
```

As shown in the Listings 2.6 there are several configuration keys for this component:

- **LOG_LEVEL**: indicates the verbosity of the log that the HM will produce during the run (possible options: DEBUG | INFO | WARNING).
- **CTRL_HOST**: Indicates the IP of the machine that will run the EM on the Control Network.
- **DATA_HOST**: The IP Address where the EM listen for the incoming data from the TCPU.
- **DATA_PORT**: The IP port where the EM will listen for the data.
- **NETWORK_THREADS**: how may threads are necessary to handle the amount of incoming data. Normally one is enough but if it is necessary to scale the system is already ready.

- **LOG_TO_SYSLOG**: This is a flag that enables the feature to log directly into the Linux SysLog daemon.
- **FILE_MAX_SIZE**: This value indicates in bytes the maximum size of a single produced file.
- **PT_FILE_PREFIX**: this is the string that will be used on the filesystem. After this prefix it will be attached a sequential number.
- **PT_FILE_POSTFIX**: this is the string that will be attached at the end of the filename that will be produced during the run.

Chapter 3

Design and implementation of the TriDAS control

This chapter presents the work done for completing the TriDAS providing the control system for all the components. The control system has been split in two parts: the TSC and the web service. Initially the GUI was only a proof of concept for showing the functionality of the web service, but at the moment, it is grown as the first use case for the TriDAS. This decision has been taken in order to logically separate the low-level control of the processes and the users.

The first part manages the execution and killing of the processes on the datacenter for starting or stopping the acquisition. Therefore this level is totally agnostic and it does not give any control about security and privileges that users can have. In an environment such as an experiment, it is obvious that a large amount of people is collaborating. So, it is necessary to implement a high level component that administrates the users and grants the privilege of controlling the TriDAS, which is a single system. This component is the web service, which is the only component allowed to issue commands to the TSC. In addition, the web service manages an escalation procedure in order to grant the permissions. This

procedure has the capability to revoke the controlling privilege implementing a "SuperUser" action that allows to solve inappropriate user's behaviors.

3.1 TriDAS Control

The *TriDAS Control* (TSC) is the software component that orchestrates all the TriDAS processes running on the data acquisition farm. The TriDAS Control implements a simple hierarchical state machine with four states, as shown in Figure 3.2:

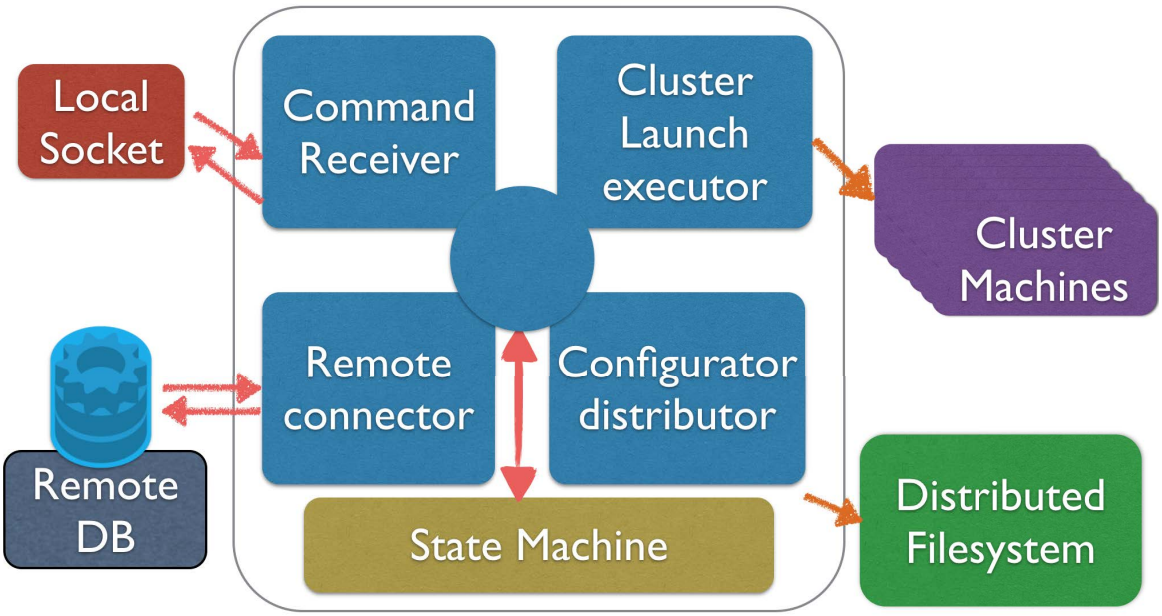


Fig. 3.1 TSC block Scheme

The TSC, moreover, computes two essential information for the launching of the entire TriDAS Acquisition, these are:

- **STARTTIME**: this information will be inserted into the final Datacard and indicates which will be the first TS that must be requested from the TSV for this acquisition.

- **RUNNUMBER**: this number identified the unique reference to this acquisition run.

The TSC is composed by several functional blocks that represent all its capabilities as shown in Figure 3.1

As the other TriDAS components the TSC has a block into the Datacard that has the content listed in the listing 3.1.

Listing 3.1 Datacard Part related to the TSC

```
1  "TSC": {
2    "DATACARD_SHAREDDIR": "/lxstorage1_home/km3/datacard/tsc"
3  },
```

- **Datacard_SHAREDDIR**: this key indicates to the TSC where is the location of the shared directory where to put the computed Datacard that will be read from every TriDAS component at launch.

Command Receiver

This block is the only entry point for the TSC, thus of the entire TriDAS. This block opens a local unix socket which permits to the TSC to communicate with only one external client at time. There is a protocol that the TSC follow in order to communicate the information to the client. The protocol is quite simple and consists in commands sent through the socket. The possible commands are:

- **status**: this command asks to the TSC in which state is the system, inquires the configured cluster and retrieves the status for the single component.
- **init “Run setup ID”**: try to enter into the “Standby” state, trying to retrieve the “Run setup” from the *ID*

- **configure**: try to enter into the “Ready” state performing the distribution of the Datacard and launching the components related to the Ready state.
- **start**: try to enter into the “Running” state trying to launch the TSV.
- **stop**: stop the acquisition and return to the “Standby” state.
- **reset**: this command will erase all the configuration, stop all the components that were running and revert the system to the “Idle” State.

For each command the TSC can answer with a *success* or an *error*.

Remote Connector

This Block has the capability to connect to a remote Database for retrieving the Runsetup configuration for an acquisition. The Runsetup as already explained is a file with a JSON content. The communication between the TSC and the remote database is based on a webAPI protocol with “GET” HTTP method.

Cluster Launch Executor

With this component the TSC can connect to each Node on the farm and launch on each computer the components as described on the Runsetup. This component uses the “libssh2” library in order to handle the ssh session natively without using external clients, providing a better error handling.

Configuration Distributor

The Configuration distributor is the functional block that will physically write the computed Datacard (the retrieved Datacard with the addition of the STARTTIME and the RUNNUMBER) into the shared directory. In this way every TriDAS component that will be launched will read the same configuration.

State Machine

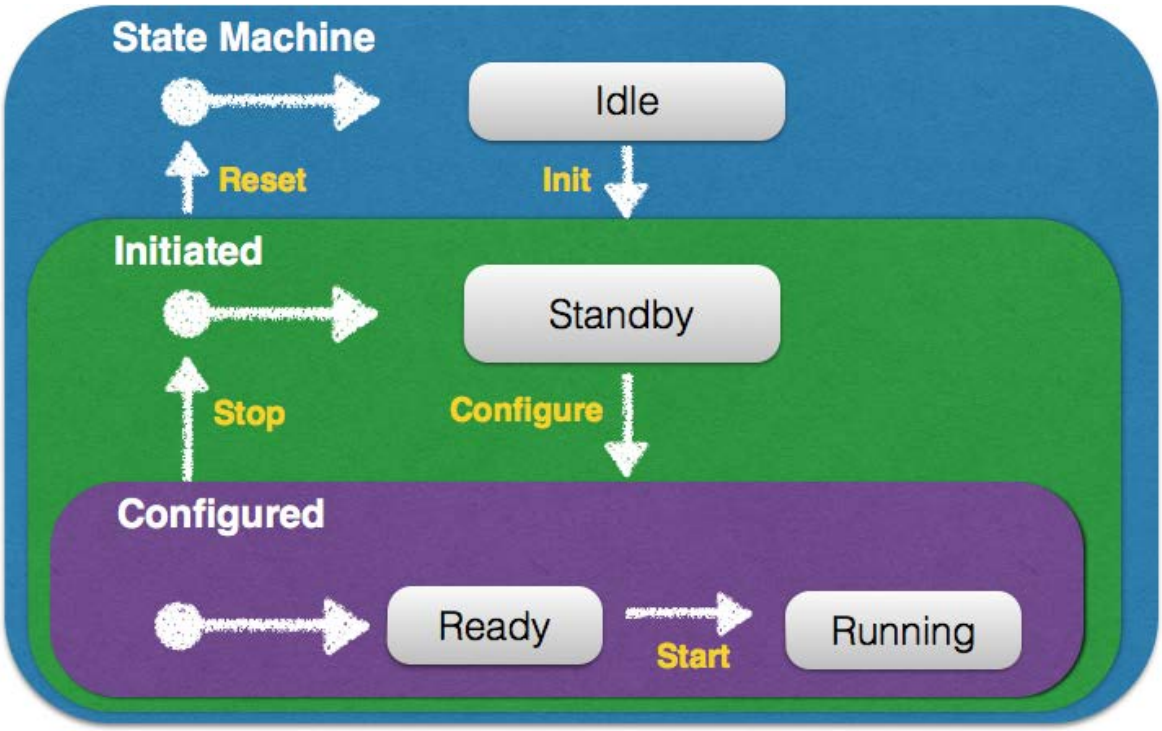


Fig. 3.2 TSC State Machine Diagram

Idle This is the initial state of the overall TriDAS state machine, where no processes are running. An *init* transition, which takes a *run setup* identifier as a parameter, executes an action that retrieves a run Datacard corresponding to the given run setup. The Datacard describes the geometry of the detector and the configuration of the TriDAS system (such as the role of each node) for this run. If the action is successful, the state machine moves into the *Initiated* sub-state machine.

Standby This is the initial state of the *Initiated* sub-state machine. Here the TriDAS Control is aware of the configuration of the TriDAS system but no processes are running yet. A *configure* transition executes an action that retrieves the *run number* and starts the Trigger CPU, HitManager and

Event Manager processes on the corresponding nodes. If all the processes start successfully the state machine moves into the *Configured* sub-state machine.

Ready This is the initial state of the *Configured* sub-state machine. Here the Trigger CPUs, the HitManagers and the Event Manager are ready to acquire, filter and save physics data coming from the FCMServers. The *start* transition executes an action that computes the start time of the run and starts the TriDAS-SuperVisor. The TriDAS-SuperVisor's role is to schedule which Trigger CPU process will compute a given TTS. The scheduling follows a credit-based mechanism to balance the load among the Trigger CPUs. If the TriDAS-SuperVisor starts successfully the data starts and the state machine moves into the *Running* state.

Running In this state the data acquisition is running.

Transitions exist to move the system back to the *Idle* and *Standby* states.

If any error occurs during a transition, the transition is aborted. Depending on the severity of the error, the system may stay in the current state or even shut-down completely.

The communication with the TriDAS Control, for example to trigger the transitions described above or to query the state of the system, is stateless and happens over an UNIX socket. Only one client can use the socket at a time.

3.2 Interface to the TriDAS Control

3.2.1 The web service

This is the real unique entry point of the entire TriDAS. This component allows users to control, inquire and monitor the acquisition system. It is a RESTFul[1] web API service, that means that it exposes command via calling HTTP methods (e.g. GET, POST) with a json payload. It is developed for allowing multiple concurrent user to see the state of the acquisition and ask for controlling the DAQ. The TSC permits only one connection to its socket and there is not any system to disconnect an idle user or broken client from that socket. With the web service via a token based authentication and an escalation algorithm is possible to have this kind of control. As shown in the Fig. **??** the web service is the only client that can issue commands to the TSC via the "Command Issuer" block. The web service is divided in two asynchronous part: the RESTFul service and the Websocket Manager (see appendix A.0.3).

The web service works with this protocol:

- A HTTP POST request arrives using the payload defined in the next chapter.
- The web service analyses the request and gives back an answer with a HTTP response.
- The web service broadcasts a message through the websocket channel.

[1] In computing, representational state transfer (REST) is the software architectural style of the World Wide Web. More precisely, REST is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. Through the application of REST architectural constraints certain architectural properties are induced: Performance, Scalability, Simplicity, Modifiability, Visibility, Portability, and Reliability.

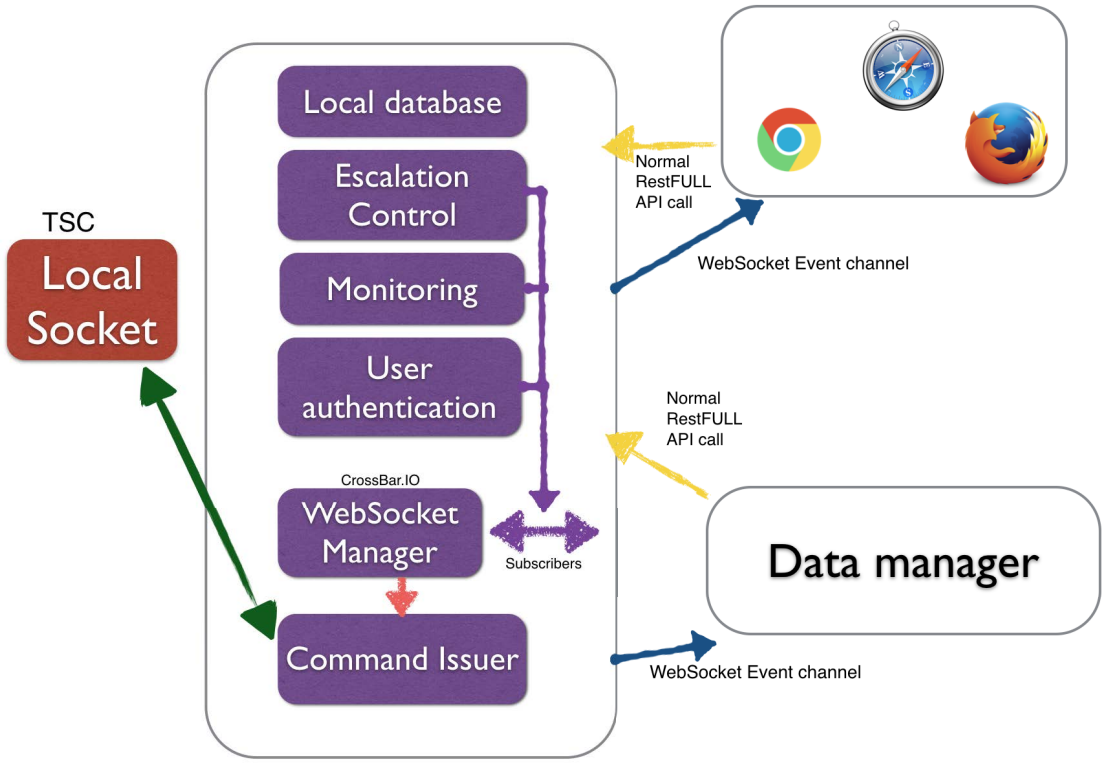


Fig. 3.3 WEBServer Block Scheme

The clients must be aware about the status of the system. The use of websockets allows to avoid a constant polling from the clients to the web service. In addition, clients can react instantaneously to the changes occurred server-side.

When a client wants to use the websocket functionality, it must implement the WAMP protocol over the websocket (see appendix A.0.3). Luckily this protocol provides libraries for a lot of programming languages[2]. If a client cannot use the websocket, the web service can be used with the standard polling mechanism.

The websocket Manager keeps alive a thread that issues the real commands to the TSC. This permits to be always responsive, even if, for example the TSC stops to answer back for any reason.

[2] http://wamp-proto.org/

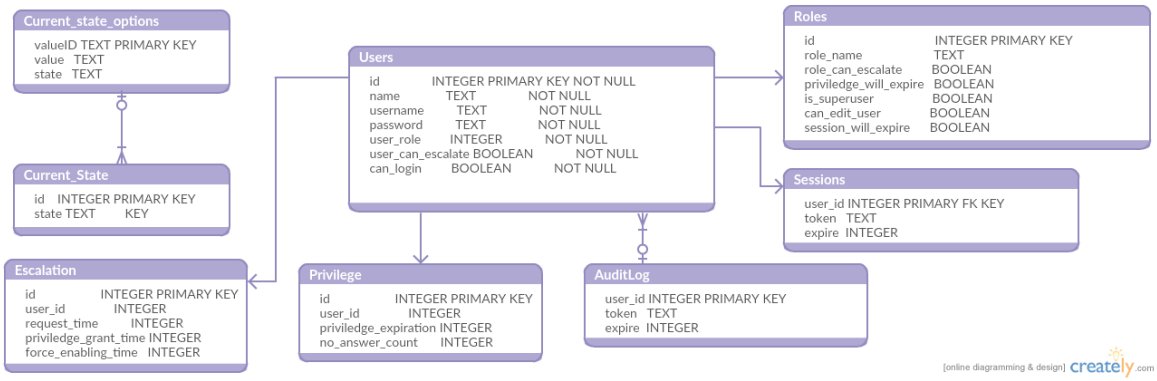


Fig. 3.4 Web Server Database Entity Relationship Schema

The peculiarity of the web service implementation is that it is very generic and it can be adapted to every system with a state machine and a single entry point. This permit to reuse the web service and transform a single user local program to a concurrent multi-user program. The web service does not mention the TSC in any part, indeed, from its point of view the TSC is only a daemon to control and inquire.

Database

The web service uses a local database for keeping information such as the users information, the Daemon Status, the escalation situation, and an audit log.

As shown in Fig. 3.4, the table *Users* is the center point of the most part of the database, actually, all the tables use the *user_id* as key for keeping information.

The password of the user is not in plain but it is a hash of a hashed password. The roles fields are simply flags that describe which permission has each role. The particular field “is_superuser_like” is for the super user Role and this flag allow an user that belong to this role to kick out an user while he is controlling the daemon. The field “can_escalate” is replicated in both the “Role” table and the “users” table in order to have more granularity on users. For example if it

is necessary to revoke the access only to a certain user for any reason without modifying his belonging role. The table session keep tracks of the life of the tokens that have been released. When a token expires, the session for that user is ended and he must do again the login.

The only two tables that are used for maintaining another kind of information are the table *Current_state* and *Current_state_options*. The first one keeps track of the current state of the daemon, the second one, instead, uses the value of the *Current_state* as key for aggregating the information that every state could provide. This two tables are materialized in order to avoid the continuous polling of the daemon.

The Table *Escalation* and *Privilege* are used by the procedure for the escalation (see section 3.2.1). This two tables keep track about the users that are trying to obtain the control of the daemon and who is controlling it at the moment.

The last table *Audit* is useful for security reason and debug reason: it makes possible to retrieve what happened to the web service and which user performed an action.

WebSocket Server

In parallel of the web service there is the WebSocket manager that routes all the messages between the web service and the remote clients, and the web service and the daemon. When a command API is called from the client, in order to perform some actions on the remote daemon, the web service will not execute that request directly but after checks if that user has: a valid session, the right permission and it is currently granted; The web service will write a message to a private “WAMP topic” that is:

- **com.tridas.execute.daemon**

There is an observer of this topic that is the **Communication Thread**. This topic has a particular configuration and only a local process, with specific credential can write on it.

Moreover, the WebSocket Server manages other topics. All the clients that would use the WebSocket features must subscribe to the following topic:

- **com.tridas.statemachine.priviledged.change**: the messages related to the privilege changing on the web service pass over this topic. E.g. when an user is escalating or an user has completed the escalation process.
- **com.tridas.statemachine.statechange**: it contains all the messages related to the state and other information that the attached daemon is providing. After a customizable time the "Communication Thread" will write in this topic the information gathered from the last action executed from the daemon.
- **com.tridas.statemachine.escalation."user released token"**: This is a special topic. When a client succeeds to login into the web service, it is provided a token that must be used in order to have the permission to execute actions. This token represent, also, a topic (built as a concatenation of *com.tridas.statemachine.escalation* + "the_token_value") on the websocket server. This topic is a private topic where there are only two actors: the web service and the specific user/client (a token is unique and known only from the logged user). When the web service must deliver a message directly to a client this topic will be used. E.g. when an user wants to escalate and obtain the control of the daemon, but there is another user that is currently privileged. In this case the web service will send over this special topic a message that will inform the current privileged user that someone wants to obtain his privilege and the protocol of "privilege exchange starts".

The “public and open to subscription” topics are read only, this means that only the service is allowed to send message over these channels.

Escalation Procedure

For allowing the users to use the daemon and permit them to issue commands a *Escalation Procedure* is needed. This procedure has the capability to assign the control of the daemon to users that request the “privilege”. This brings some problems during the managing of the escalation but is possible to identify different scenario to be taken into account:

- *There is not any privileged user*: this is the simplest scenario; if the system is in this situation the privilege will be granted to the first user who ask for it. The grant is given for a certain amount of time, when it expires the privilege is automatically revoked.
- *There is another privilege user*: in this case a user has requested and obtained the privilege to control the daemon and another user wants it. There is a collision so the system tries to resolve it with a request to the current user to release the privilege. If the clients are using the websocket a message on the “private channel” (topic with the token) of each user is sent. Otherwise, it is the client itself that must poll the server in order to know if it is losing the privilege. The privileged user must answer. If he will not answer in a certain amount of time (two minutes at the moment), he looses the privilege automatically and the system grants the privilege to the other user. If the current privileged user replies, the answers can be only two:
  - *yes, “release my privilege”*: in this case the solution for the system is easy, the web service will revoke the privilege from the current user

and inform the other that from this moment he is the current privileged (Fig. 3.5).

– *no, "I want to keep it"*: when this happen the system will inform the other user that his try has failed but the system count how many times the negative answer is given. After two consecutive negative answer given and a given idle time the system enables the "forced escalation" procedure. This grant the privilege to the requesting user without any confirmation from the current user. There is a particular kind of users that has the flag "is_super_user" set, that have always the possibility to use the "forced escalation". This flag is normally used for administrative user that need the control for fixing problems (Fig. 3.6).

This procedure needs a synchronization among concurrent users and this is provided by the database using transactions.

Communication Thread

This component is the actual communicator with the daemon, this thread opens the connection to the daemon and subscribes itself as an observer of the topic:

- **com.tridas.execute.daemon**

when a command is received over this topic the thread will execute the corresponding action and after that it consolidates the information on the database and writes a message on the topic :

- **com.tridas.statemachine.statechange**

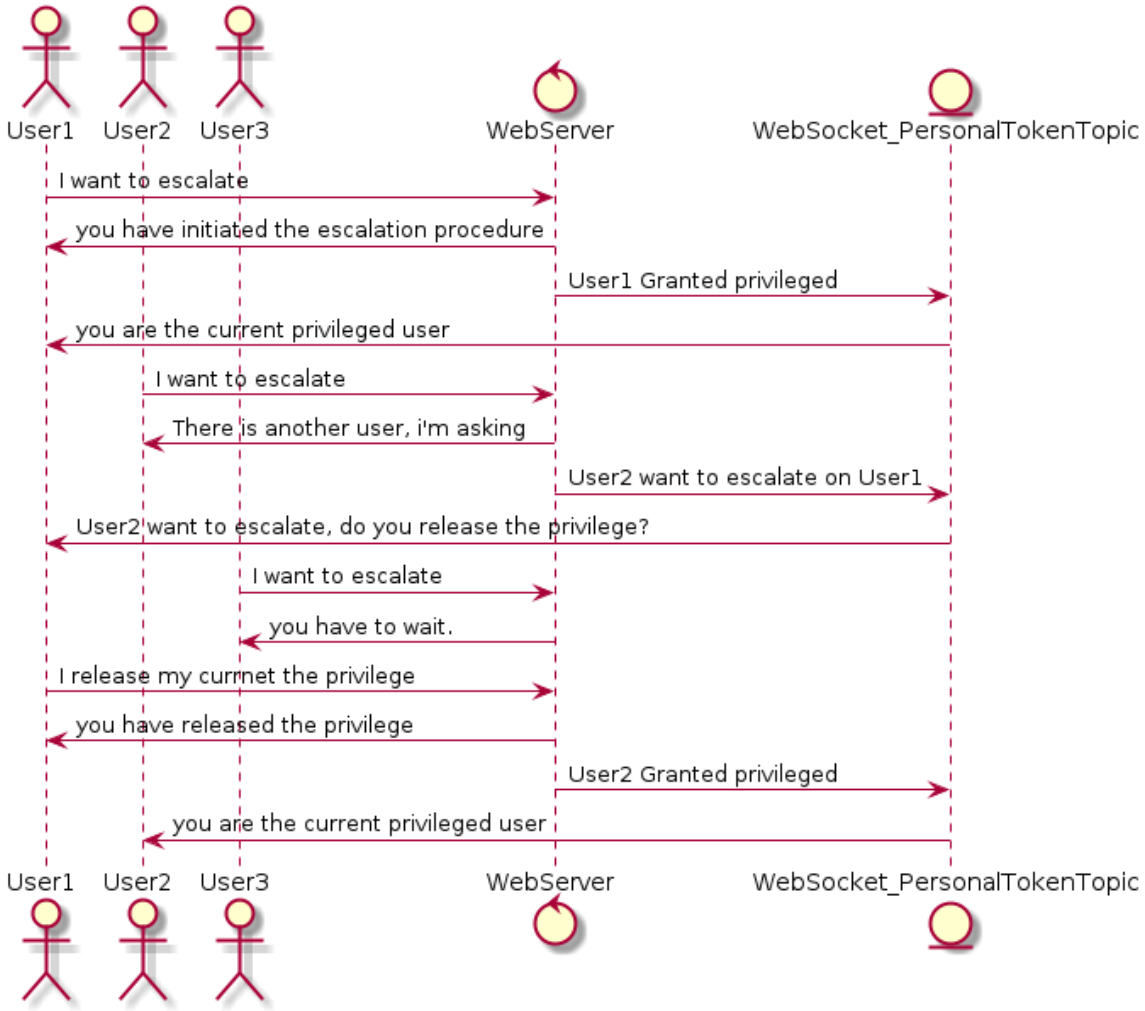


Fig. 3.5 Escalation Procedure with an yes answer sequential diagram

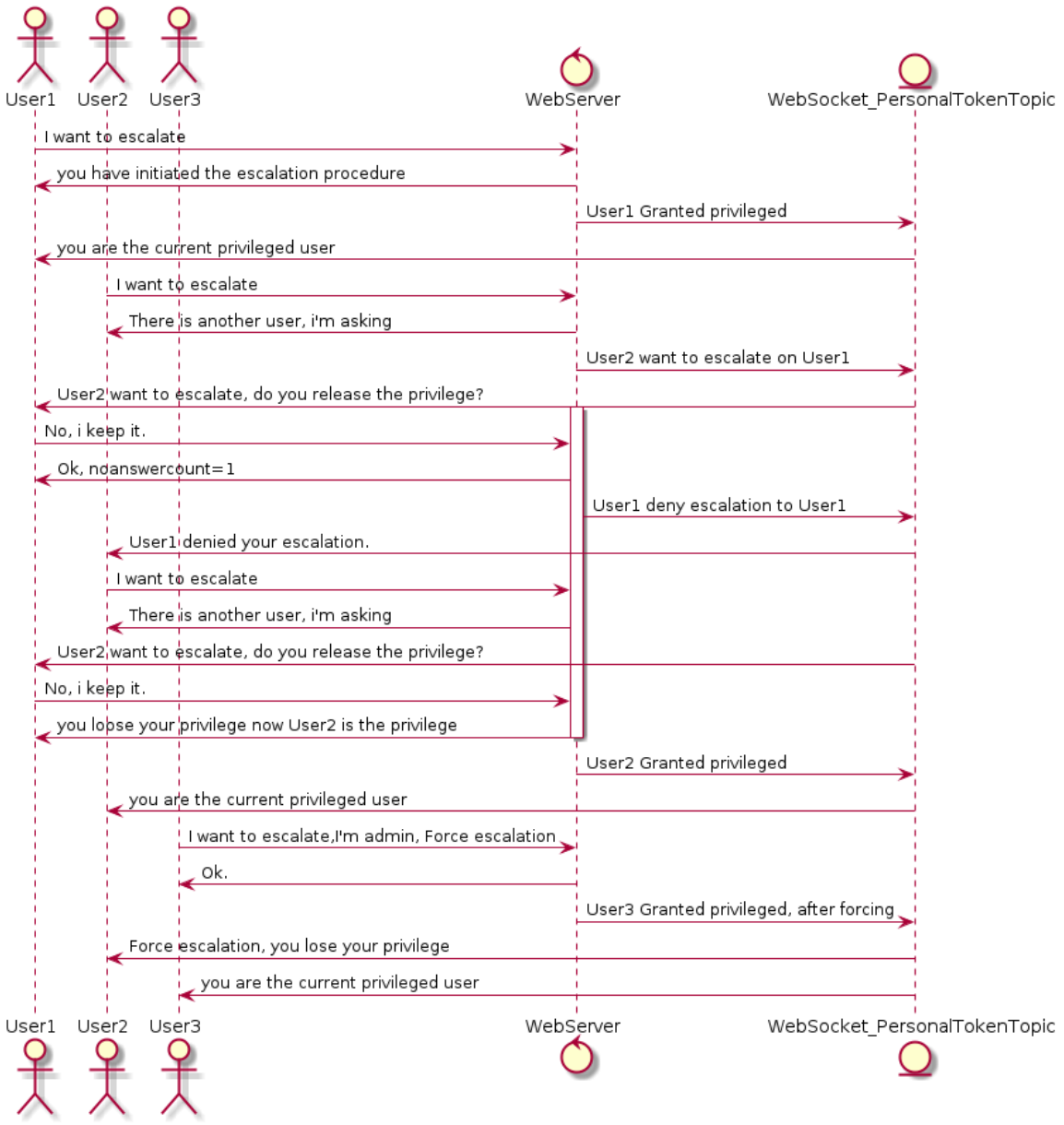


Fig. 3.6 Escalation Procedure sequential diagram with a “no” answer and a forced procedure

3.2.2 The APIs exposed by the web service

The web service provides APIs for allowing the users to have access to the daemon. In the next sections, all the available APIs are described.

Login API

In order to access to the service, it is necessary to call this API at the beginning of the session.

API url: https://web service.Domain/login

This API requires the parameters listed in Listing 3.2.

POST data

Listing 3.2 Login Post Data

```
{
    username: "[username]",
    password: "[password]",
}
```

If the login is requested by the“datamanager[3]” user, the web service does not limit the duration of his session. The password field must contain a sha256 hash of the true password value.

The web service will answer as shown in the Listing 3.3.

Answers data

Listing 3.3 Login Answer Data Example

```
{
    role: "admin",
```

[3]The datamanager is another slow control that during the first phase of the experiment represented the slow control of the acquisition

```
    token: "cfc32ff100e72e159b83817f4cd56481",
    action: "ok",
    message: "User matteo succesfully logged in"
}
```

For normal users the session has a duration of 15 minutes. Every time a user contacts an API the session is automatically renewed.

If any other APIs is called without first calling this API, the web service will always reply with the HTTP error **403 Forbidden**.

check Token API

This API permits the clients to use the last obtained token as login, instead to call the "login API". If the token is not expired it will be renewed. **API url**: https://web service.Domain/verifyToken

POST data:

Listing 3.4 Login via auth token Example

```
{
    authToken: "token"
}
```

Answers data:

Listing 3.5 Login via auth token Answer

```
{
    role: "admin",
    action: "ok",
    message: "User matteo successfully logged in"
}
```

Logout API

This API permits to explicitly destroy an user session on the server.

API url: https://web service.Domain/logout

POST data:

Listing 3.6 Logout via auth token

```
{
    authToken: "token",
}
```

Answers data:

Listing 3.7 Logout via auth token Answer

```
{

    action: "ok",
    message: "User matteo successfully logged out"
}
```

Command API

In order to control the Daemon is necessary to use this API. By design the daemon can accept command only from an “escalated” user. If a command is issued from a user that is authenticated but has not completed the “Escalation Procedure” the web service will always answer with the HTTP error **403 Forbidden**.

API url: https://web service.Domain/commands

POST data:

Listing 3.8 Command Post data

```
{
    command: "[init|configure|start|stop|reset]",
    param: "[better explained afterwords]",
    authToken: "[previously received token]"
}
```

The *param* key is required with the command **init** and the request is:

Listing 3.9 Command Init example

```
{
    command: "init",
    param: "[runsetup]",
    authToken: "[previously received token]"
}
```

In all the other cases the *param* key will be ignored. When the request is recognized and it is allowed, the web service will try to issue the request to the back-end in order to generate an answer. The answers can be: success or fail:

Listing 3.10 Command Api answer example

```
{
    action: "ok|fail",
    currentState: "idle|stanby|ready|running"
}
```

Status API

This API can be used from any authenticated user. It is important to underline that at every state-machine event a websocket notification is sent to the client in order to minimize the requests to the server. The behavior of the client is left to the developer decision.

API url: https://web service.Domain/status

POST data:

Listing 3.11 Status Api example

```
{
    command: "state",
    authToken: "[previously received token]"
}
```

The answer and the notification have the same structure and content.

Listing 3.12 Status Api answer example

```
{
    currentState: "idle|stanby|ready|running"
    Processes: {
        HM: "[n_of_processes]",
        TSV: "[n_of_processes]",
        TCPU: "[n_of_processes]",
        EM: "[n_of_processes]"
        },
    Other: "{ [better explain afterwords] }"
}
```

Inside the *Other* key there are all the other information that the daemon is giving but that are not related to the processes.

RunSetup API

The RunSetup API is provided in order to allow a client to know the list of the avaiable RunSetup on the remote DB.

API url: https://web service.Domain/runsetup

POST data:

Listing 3.13 Runsetup Post Data

```
{
    command: "getRunSetupList",
    param: "",
    authToken: "[previously received token]"
}
```

Answer:

Listing 3.14 Runsetup Answer example

```
{
    {
        id: "ID",
        name: "[RunSetup Name]",
        othersKeys: "[usefull information]"
    },
    {
        id: "ID",
        name: "[RunSetup Name]",
```

```
        othersKeys: "[usefull information]"
    }
}
```

Escalate API

In this section, the set of commands provided by the web service in order to escalate privileges will be described.

API url: https://web service.Domain/escalate

This API has several commands, as in the following.

- **"Command: Am I Privileged?"**

  This command permit to know in every moment if the current user is allowed to send commands to the Daemon. If a "No answer" (false) is returned, any attempts to use the "Command API" will then cause an HTTP error: **403 Forbidden**.

  POST data:

Listing 3.15 Am I Privileged data

```
{
    command: "amiprivileged",
    authToken: "[previously received token]"
}
```

Answers data:

Listing 3.16 Am I Privileged answer example

```
{
    result: "true|false",
```

```
    currentPriviledgedUserName: "username | '' ",
    currentPriviledgedName: "name | '' "
    currentEscalatingName: "name | '' "
    message: "the current privileged user is  XX | there aren'
       t any priviledged
    user" + " user XX is escalating" |  " "
    priviledgeWillExpireInSeconds: "seconds to priviledge
       exipration"
}
```

- **"Command: I would like to escalate".**

  This command is needed if a client wants to become a Privileged users.

  POST data:

Listing 3.17 I would like to escalate example

```
{
    command: "iwouldliketoescalate",
    forceEscalation: "true|false"
    authToken: "[previously received token]"
}
```

The key *forceEscalation* is dangerous to use. With a true value, it forces the web service to release the priviledge from the current privileged user to the issuing user. This key is thought in order to avoid a buggy client to always answer "no" to the service "Authorize escalation". This parameter is simply ignored until a client is answering "no" to the service "Authorize escalation" for more that 2 minutes. After that time every new request with this

parameter set will be evaluated. When the procedure is started a message will be sent to the notification channel identified by: “com.TriDAS.escalation” . “token” . of the current privileged user.

Answers data:

Listing 3.18 I would like to escalate answer example

```
{
    result:
    "procedureinitiated|procedureAlreadystarted|WAFOU|WAFOUFEE
       |escalationCompleted|youAreAlreadyPriviledged",
    currentPriviledgedName: "[name of the current privileged
       user]",
    currentEscalatingName: "[name of the current escalating
       user]",
    message: "[other information eg The procedure is started
       by you wait, The
    user XX has already started the procedure, ]",
    timestamp: "[time stamp of the begining of the procedure]"
       ,
    secondsToForceEnabling:"seconds",
}
```

The meaning of the answers is:

- **procedureinitiated**: this client is the initializer of the escalation procedure, if it will end correctly, the client will become a privileged user.

- **procedureAlreadystarted**: the procedure is on going by another user and this client has to wait.
- **WAFOU**: (Waiting Answer From Other User) this client has already asked to become a privileged user and now it has to wait that the procedure completes. The maximum time of completion could be 2 minutes or 5 minutes depending on the behavior of the other users.
- **WAFOUFEE**: (Waiting Answer From Other User, Force Escalation Enabled) the privileged user is continually answering "no" so from now the parameter "forceEscalation" is taken into account.
- **escalationCompleted**: There was not any privileged user, so now the client has automatically became the new privileged user
- **youAreAlreadyPriviledged**: this client is escalating on itself, nothing will be done.

Everytime something happens server-side, a notification will be sent on the topic:"com.TriDAS.statemachine.priviledged.change" with the following content:

Listing 3.19 push answer example

```
{
    currentEscalatingName: "",
    currentPriviledgedName: "Matteo Favaro",
    currentPriviledgedUserName: "matteo",
    priviledgeWillExpireInSeconds: 900
}
```

- **Command: Am I losing the privilege"?**

This command is provided in order to know if someone has started the "Escalation procedure". When the procedure is started this is the way to know who has started the procedure.

POST data:

Listing 3.20 Command: Am I losing the privilege example

```
{
    command: "imlosingprivilege",
    authToken: "[previously received token]"
}
```

Answers data:

Listing 3.21 Status Api answer example

```
{
    result: "true|false",
    message: "[some useful description eg. User XX wants to
       became
    privileged,
    contact answer service]"
}
```

- **"Command: Authorize escalation"**

  When "Am I losing the privilege" API returns true, the client must call this API within 2 minutes. If the call does not happen, the web service assumes a positive answer. If the current privileged user repeatedly answers "no" for 5 minutes, the web service will give to the requesting user the possibility to force the privilege escalation.

POST data:

Listing 3.22 Status Api answer example

```
{
    command: "authorizeescalation",
    authorize: "[yes|no]",
    message: "[reason mandatory]",
    authToken: "[previously received token]"
}
```

Answers data:

Listing 3.23 Status Api answer example

```
{
    result: "ok",
}
```

- **”Command: Release Privilege”**

  This command explicitly release the privilege acquired.

  POST data:

Listing 3.24 Status Api answer example

```
{
    command: "releaseprivilege",
    authToken: "[previously received token]"
}
```

Answers data:

Listing 3.25 Status Api answer example

```
{
    result: "ok",
}
```

Escalation Procedure Best Practice

In this section, the order and how to contact the provided “escalation” commands are described.

1. *(optional but useful depending on developer implementation)* before contacting the command API it is a good practice to contact the escalate API with the *amiprivileged* command:
   (a) If a “yes” answer is returned, the user is already privileged and can send commands and control the data acquisition.
   (b) If a “no” is returned, continue with the procedure.
2. If the user wants to escalate and gain the power to control the daemon, it is needed to call the *iwouldliketoescalate* command. The next steps depends on the implementation of the client; it can either poll the API with this command or with the *amiprivileged* command. Moreover, if the client is using the websocket feature, it just need to wait for a message on the token topic.
3. When a client is privileged, it must keep polling the escalate API with the *imlosingprivilege* command, unless the websocket feature is active.
4. When a client discovers that it is losing the privilege, it must contact the escalate API with the *authorizeescalation* command.

5. It is a good practice to release the privilege contacting the escalate API with the *releaseprivilege* command.

3.3 A graphical Gui

A graphical application has been made for having the possibility to use the TriDAS.

3.3.1 Purpose

This application has been designed for several reasons, one of them is the administration of the users and, of course, the control of the acquisition.

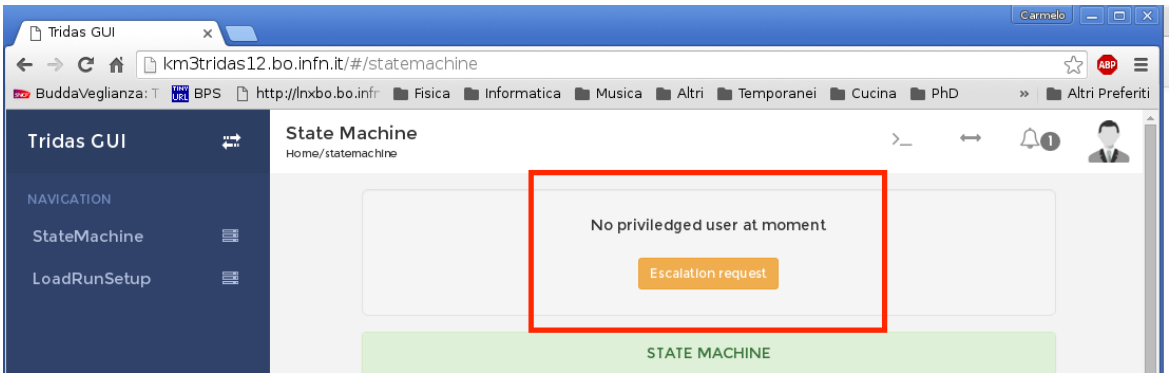


Fig. 3.7 A representation of an running acquisition on the GUI without the priviledge

In Fig. 3.7 and Fig. 3.8 it is possible to notice how the privilege mechanism is notified. The status of the user privilege is always shown with a clear indication about the remaining time.

In Fig. 3.9 is shown a special feature of the GUI: the upload of a local runsetup to the system in order to make it usable from the TSC.

In Fig. 3.10 are shown different things:

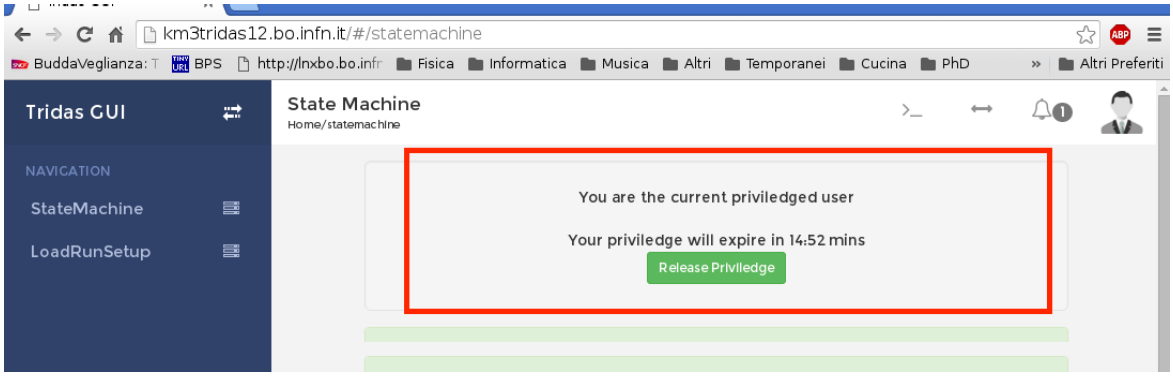


Fig. 3.8 A representation of an running acquisition on the GUI with a gained priviledge

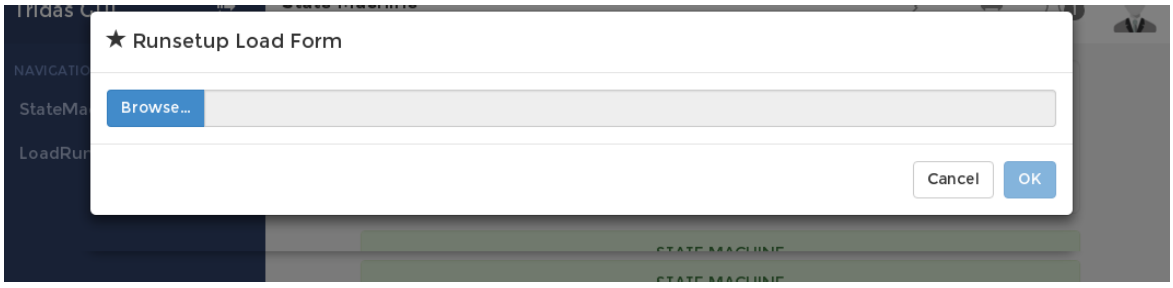


Fig. 3.9 An example of other purposes of the GUI: the upload of the Runsetup from a local client.

1. The status icon: there are two icons that indicate respectively the the status of the daemon (1) and the status of the websocket connection (2).

2. The notification Panel: the system uses the websocket features and in this GUI they are implemented as notifications that appear when messages are sent on the websocket channel(3).

3. The personal user Manu: Each user can view the personal data such as the role.

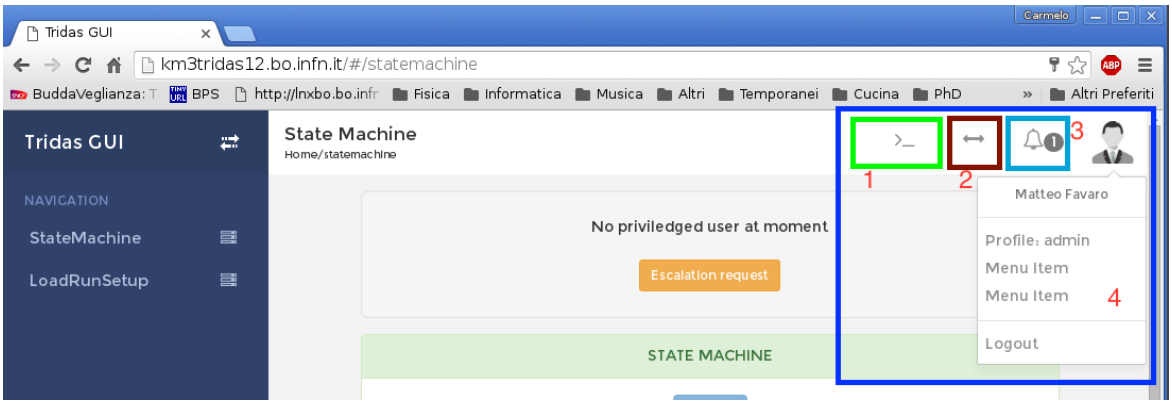


Fig. 3.10 The GUI uses HTML 5 technique and AngularJS: with dinamic icon, dynamic notification

Finally, in Fig 3.11 is represented a typical use case from the user's point of view. The users connect to the web service via the standard HTTP channel and at the same time they connect to the websocket server for receiving the notification through the various topics. The web service is connected to the web socket server via a private channel that only the web service can use. The websocket server (Crossbar.io) keeps alive the communication thread that is continuously connected with the TSC, the communication thread receives notification to execute command from the web service and notifies the topic about the results.

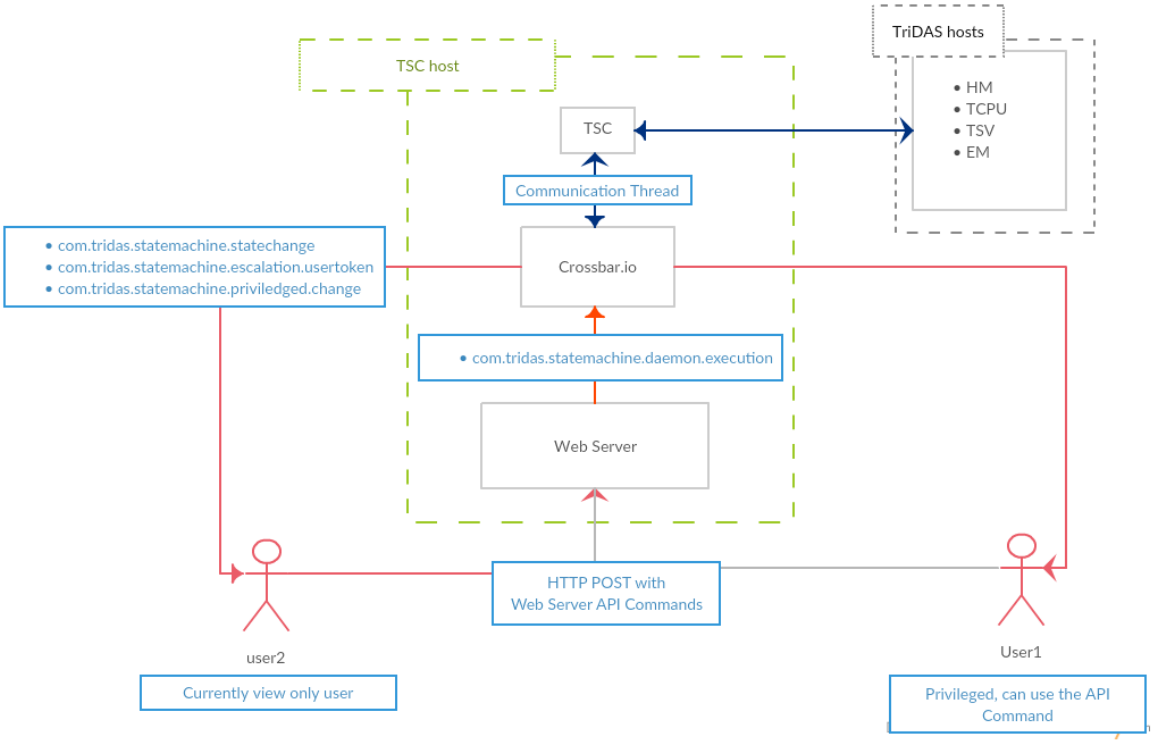


Fig. 3.11 Representation of the connections status between all systems: two users, one privileged and one not, issue command via HTTP POST and receive notification from the WebSocket Server via the subscribed topic; the web serve issue the command to the communication thread, that is kept alive from crossbar.io. The communication Thread will communicate with the TSC.

Chapter 4

Tests

In this chapter, a relevant test and its results about the TriDAS system will be presented.

The aim of this test is to observe how the system behave changing the load (i.e. the number of towers to be processed). The environment configuration and the number of nodes, their role and processes are constant during the test. The used machines are the same as those available in the Portopalo's datacenter.

In the test bench there are available machines for simulating at most 4 towers. So the test can only push until that limit.

4.0.1 The farm

The machines that have been used are connected as shown in Fig. 4.1, and they have the following specifications:

- **TCPU machines**: 4 nodes
  - *Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz*, 32 cores
  - *32 GB of ram*

  - *1 Gb connection for control network*
  - *1 Gb connection for data from TCPU to EM network*
  - *10 Gb connection for data from HM to TCPU network*

- **HM machines**: 4 nodes
  - *Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz,* 8 cores
  - *128 GB of ram*
  - *1 Gb connection for control network*
  - *1 Gb connection for data from FCM to HM network*
  - *10 Gb connection for data from HM to TCPU network*

- **TSV machine**: 1 node
  - *Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz,* 8 cores
  - *16 GB of ram*
  - *1 Gb connection for control network*
  - *1 Gb connection for token network*

- **TSC, Web server, GUI machine**: 1 node
  - *Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz,* 8 cores
  - *16 GB of ram*
  - *1 Gb connection for control network*

- **EM**: 1 node
  - *Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz,* 8 cores
  - *16 GB of ram*

- *1 Gb connection for control network*

- **FCM**: 6 node
  - *Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz*, 8 cores
  - *16 GB of ram*
  - *1 Gb connection for control network*

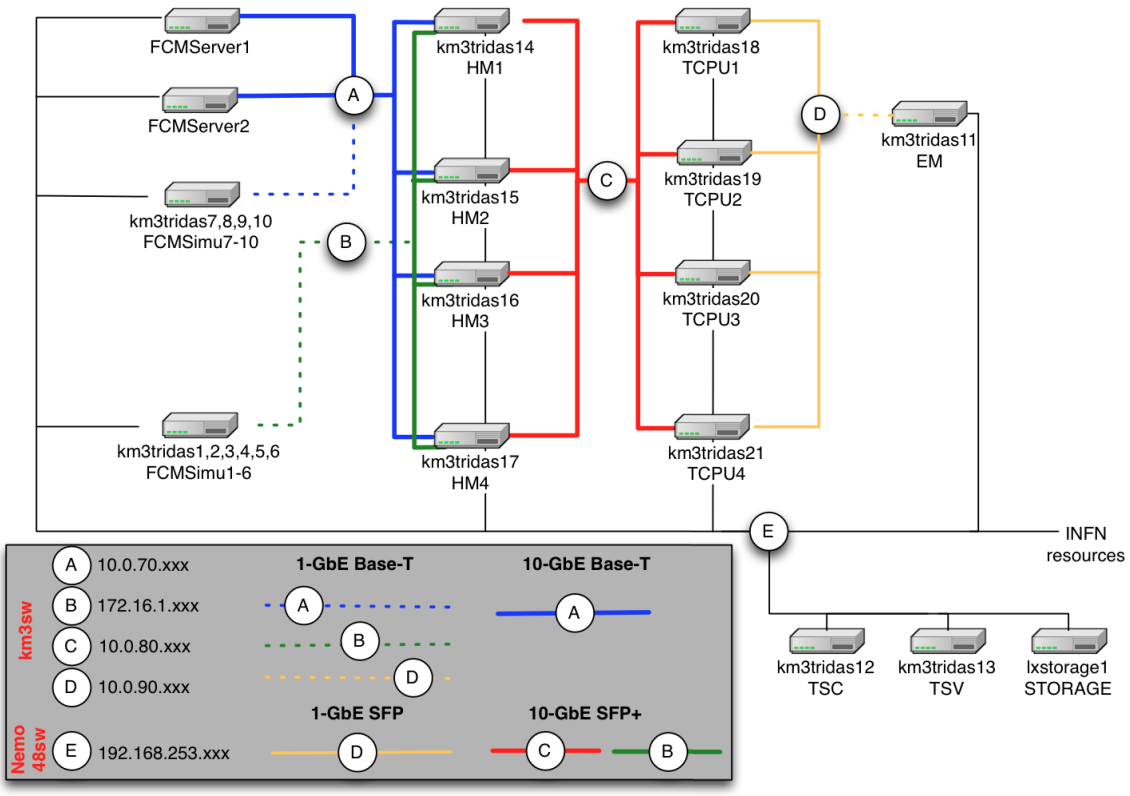


Fig. 4.1 The TriDAS core network topology

4.0.2 Configuration

For this test the farm has been configured as follow:

- 4 TCPU one for each available machine, each TCPU process will elaborate 24 parallel TTS at time for a total of 96 parallel TTS at time.

- 4 HM one for each available machine, each HM will be elaborate an entire towers at time connecting to 7 different FCM processes.
- 1 TSV, 1 EM, 1 TSC, web server and GUI, each process are located at the machine as described in previous section.
- 56 FCM process, the processes are equal distributed over 6 machine, in each machine there are 10 FCM process.

It has been decided to set 24 threads (1 TTS per thread so 24 parallel TTS) because the machine has 32 cores but there are other threads into the TCPU process, for example the connection thread and the queue handling, it has been decided to leave some cores for these threads.

The number of FCM processes per machine is determined by the frequency of event generation. Indeed, a FCM process that generates events at 50KHz is producing data at 5.5Mbs. The available bandwidth related to the data network of a FCM server is 100Mbps (nominally), so 10 FCM processes on the same machine generate 55Mbps.

Eight different runs have been performed, each with an increment of half tower (7 floors) with respect to the previous. Each run lasted for 30 minutes.

At the end of each run, the logs of the TCPU have been collected and pushed in a git repository with a continuous integration job that processes data and creates the graphs.

4.0.3 Results

In Fig. 4.2 the results of this test are summarized. As the load increases, the more is the time spent by the L1 trigger algorithms. With 4 towers the maximum value that has been recorded is 1,6s.

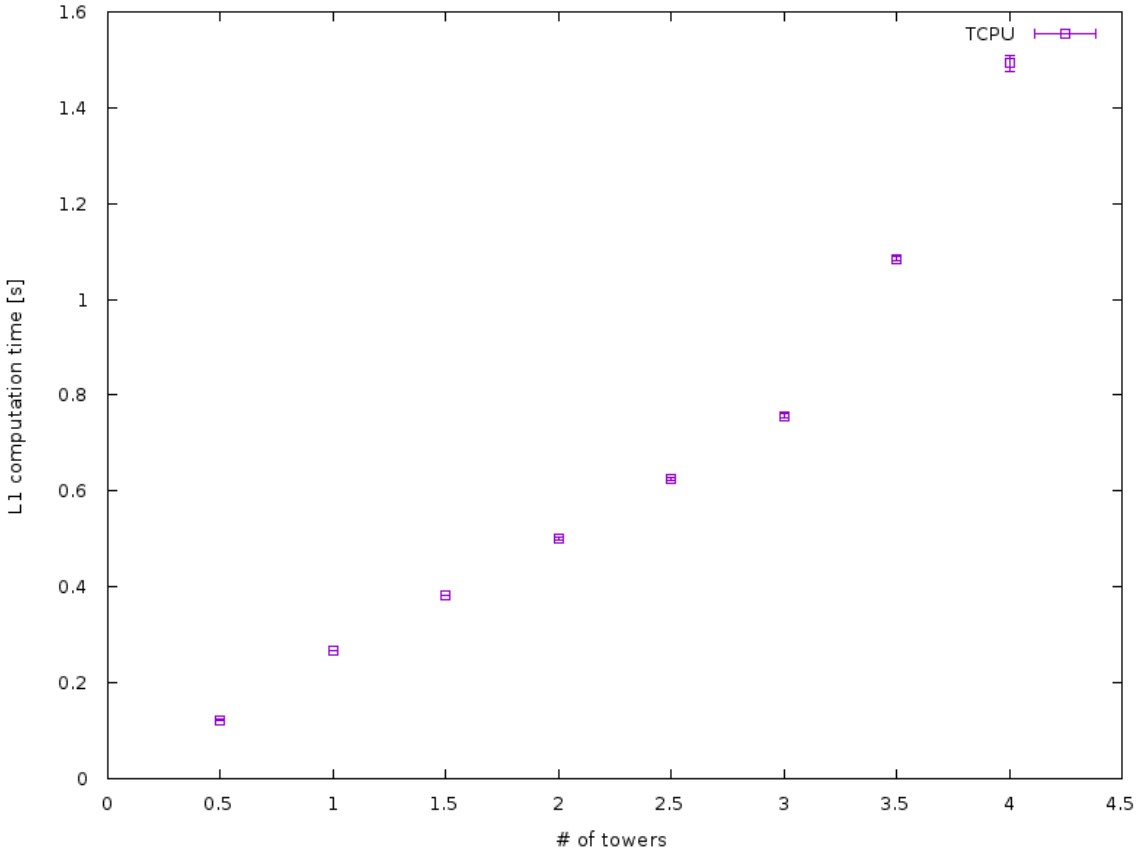


Fig. 4.2 Represetation of the elaboration of the L1 Time execution

For better understand the meaning of this value, it is necessary to put some upper-limit to the TTS processing time, above which the data acquisition will result out of sync with the data produced from the FCM server. The HMs slice the data in 200ms-long slots:

$$time_{spent\_per\_TTS} < 200ms$$

In the TriDAS's case the *time_spent_per_TTS* is proportional to the number of available parallel TTS, so:

$$time_{spent\_per\_one\_TTS} = time\_window * n\_parallel\_TTS$$

Thus, until the *time_spent_per_TTS* is less than the time window the acquisition and elaboration are synchronized.

Applying our configuration data to the formula above the time per TTS becomes:

$$time_{spent\_per\_one\_TTS} = 200ms * 96\ (parallel\ TTS)\ time_{spent\_per\_one\_TTS} = 19,2s$$

This means that the every TCPU has at most 19,2s for completing a TTS.

Looking at the results, the maximum time of 1,6s that has been recorded is well below the upper limit calculated above.

It could be more accurately it is possible to describe how much time the TCPUs have spent to elaborate each TTS:

$$time_{one\_single\_TTS\_over\_entire\_TriDAS} = 1,6s\ /\ 96 = 0,01\bar{6}s \approx 17ms$$

$$17ms \ll 200ms$$

This means that more than 17 seconds are available to the processing of the L2 triggers. It is important to note that as more time is available, as more complex the algorithms running during the data acquisition could be. TriDAS

makes possible to increase this time by simply increasing the number of parallel TTS. At Portopalo's farm there are 48 machines of TCPU class, this means that the parallel TTS can become:

$$48 * 24\ (pTTS) = 1152$$

so the total available computational time becomes:

$$200ms * 1152 = 230,4s$$

This result is very encouraging and at the moment it is possible to state that the system is ready to support the acquisition from at least 4 towers.

More investigations will be carried out, indeed we are waiting for an expansion of the test bench in order to be able to simulate the foreseen 8 towers.

Chapter 5

Conclusion

The TriDAS has been improved to sustain the foreseen 8 towers detector. Its performances and scalability are under intense test, with long duration runs and varying the incoming throughput using either real FCMServer and simulation programs, using a test bench that reproduces a scale of the real farm in Portopalo. New trigger algorithms are under development serving different kinds of physics analysis, e.g. multi-messenger external alerts, high energy neutrino induced showers and astrophysical source detection.

The test phase demonstrates that the system is stable and the users are able to control it properly. Moreover, in November 2015 the installation and functionality test of the farm in Portopalo has been completed. Extended tests of the TriDAS will be also realized in the Portopalo infrastructure, in advance with respect to the first deployment of the Towers. This will make possible tests in the real situation, exploiting the actual facility available in the shore station. In addition to that, increasing the computing resources with respect to what available in the test-bench, more realistic results will be achieved.

Moreover the whole system is using State-of-the-art of latest technologies. Such as the HTML5, websocket, $\emptyset MQ$ etc. In particular the web service is

written for being adjustable to different systems and applications. Indeed with limited modifications it can control and export web API for every single-user local-program, that uses some kind of local communication, such as socket, pipe or similar. With this new approach it is possible to reuse the web service as a “hat” and so improve the development time of new applications by having a control system already ready.

At the same time, the whole TriDAS system is modular. This permit to change the environment and the purpose with small efforts, moreover, this permit to be more agile on DAQ development, avoiding to re-implement large part of the code, concentrating on more important things such as the trigger algorithms.

In the near future this system will be used in real-time data acquisition with the deployment of the first tower in Portopalo. There is a lot of excitement around this event because the system will be used in “the real world” for the first time. For this reason the TriDAS is continuing a deep testing phase in order to be sure to successfully harvest the data from the undersea towers.

During the development phase we used many tools and programming methods such as git, agile, TDD, continuous integration. This has permitted to be very fast and precise. For this reason the test is confirming that the whole system at the deployment will be ready and reliable.

References

[1] K. Collaboration, “Km3net web site,” http://www.km3net.org/home.php, 2015, [Online; accessed 11-Dec-2015].

[2] S. Aiello et al., “Measurement of the atmospheric muon depth intensity relation with the {NEMO} phase-2 tower,” *Astroparticle Physics*, vol. 66, pp. 1 – 7, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0927650514001960

[3] M. S. T. Chiarusi, “High-energy astrophysics with neutrino telescopes,” *European Physical Journal*, vol. 65, no. 649, p. 3, 2010.

[4] C. Pellegrino, et al., “The trigger and data acquisition for the NEMO-Phase 2 tower,” *AIP Conference Proceedings*, vol. 1630, no. 158, p. 7, 2014.

[5] M. de Jong, “KM3NeT: The next generation neutrino telescope,” *International Cosmic Ray Conference (33 ; 2013 ; Rio de Janeiro)*, vol. 3, no. 0891, p. 4, 2013.

[6] Markov, M. A., “Proceedings int. conf. on high energy physics,” *Proceedings Int. Conf. on High Energy Physics*, p. 183, 1960.

[7] Pellegriti, M. G., et all, “Long-term optical background measurements in the capo passero deep-sea site,” *AIP Conference Proceedings*, vol. 1630, no. 1, pp. 94–97, 2014. [Online]. Available: http://scitation.aip.org/content/aip/proceeding/aipcp/10.1063/1.4902780

[8] B. Bakker, “Trigger studies for the Antares and KM3NeT neutrino telescopes,” *Bachelor Thesis*, p. 314, 2011.

[9] A. M. M. Spurio, Tech. Rep.

[10] M. Manzali, T. Chiarusi, M. Favaro, F. Giacomini, A. Margiotta, and C. Pellegrino, “The trigger and data acquisition system for the 8 tower subsystem of the {KM3NeT} detector,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900215014266

[11] A. L. et al., “Nanet: a configurable nic bridging the gap between hpc and real-time hep gpu computing,” *Journal of Instrumentation*, vol. 10, no. 04, p. C04011, 2015. [Online]. Available: http://stacks.iop.org/1748-0221/10/i=04/a=C04011

[12] J. Brunner, “General purpose data format for antares simulation and reconstruction, antares internal note,” CCPM, Tech. Rep. ANTARES-Soft/1998-007, 1998.

[13] ——, “Updated tag list for new antares event format, antares internal note,” CCPM, Tech. Rep. ANTARES-Soft/1999-003, 1999.

[14] B. Comunity, “Boost web site,” http://boost.org, 2015, [Online; accessed 01-Gen-2016].

Appendix A

Development Tools

A.0.1 Boost

The Boost C++ Libraries[14] are a collection of libraries based on the C++ standard. The License that this tool uses is the Boost Software License, which allows anyone to use, modify, and distribute the libraries for free. The libraries are platform independent and support most popular compilers, as well as many that are less well known.

The Boost libraries are developed and published with the contributions of the boost community. The community is composed by a large group of C++ developers from around the world coordinated through the web site *www.boost.org* as well as several mailing lists. The library are stored on a GitHub Repository freely accessible by every one. The target of the community is to develop and collect high-quality libraries, complementary to the standard library. The libraries that during the development process had proved valuable and become important for the development of C++ applications, have a good chance of being included in the standard library.

The Boost community emerged around 1998, when the first version of the standard was released. Now it plays a big role in the standardization of C++.

There is no relationship between the Boost community and the standardization committee. The developers are often involved in both groups.

The current version of the C++ standard, approved in 2011, includes libraries that have their roots in the Boost community.

The use of Boost libraries has often proved useful for increasing productivity in C++ projects, especially when your requirements go beyond what is available in the standard library. This happens because the Boost libraries evolve faster than the standard library. Thus, developers can benefit from progress made in the evolution of C++ more rapidly.

A.0.2 ZMQ

ØMQ was born in 2007 as an iMatix project to build a low-latency version of OpenAMQ messaging product, with Cisco and Intel as partners. Since the beginning, *ØMQ*'s target was to get the best performance possible out of hardware. To achieve this, the libraries are developed in multithreading which is also the key feature of this libraries.

Afterwards, a technical white paper was published which says:

"Single threaded processing is dramatically faster when compared to multi-threaded processing, because it involves no context switching and synchronisation/locking. To take advantage of multi-core boxes, we should run one single-threaded instance of an AMQP implementation on each processor core. Individual instances are tightly bound to the particular core, thus running with almost no context switches."

ØMQ is popular for several reasons:

It is open source and it is supported by a large community. It includes an ultra-simple API based on BSD sockets. This API is familiar, easy to learn, and conceptually identical no matter what is the language. It implements real messaging patterns like topic pub-sub, workload distribution, and request-response. This means $\emptyset MQ$ can solve cases for connecting applications. It seems to work with every programming language, operating system, and hardware. It provides a single consistent model for all language APIs. This means that the investment in learning $\emptyset MQ$ is rapidly portable to other projects. It is licensed as LGPL code. This makes it usable, with no licensing issues, in closed-source as well as free and open source applications. It is designed as a library that is linked with the applications. This means there are no brokers to have to be started and managed: the less are the moving pieces, the less they can break or go wrong. Above all, it is simple to learn and use. The learning curve for $\emptyset MQ$ is roughly one hour. And it has odd uses thanks to its tiny CPU footprint. As Erich Heine writes, “the [$\emptyset MQ$] performance tests are the only way we have found yet which reliably fills a network pipe without also making cpu usage go to 100

Most $\emptyset MQ$ users come for the messaging and stay for the easy multithreading. No matter whether their language has multithreading support or not, they get perfect scaling to any number of cores, or boxes. Even in Cobol.

One goal for $\emptyset MQ$ is to get these "sockets on steroids" integrated into the Linux kernel itself. This would mean that $\emptyset MQ$ disappears as a separate technology. The developer sets a socket option and the socket becomes a message publisher or consumer, and the code becomes multithreaded, with no additional work.

A.0.3 CrossBar.IO

Crossbar.io is an open source unified application router implementing the WAMP protocol, an open standard WebSocket subprotocol.

Crossbar.io directs and transmits messages between these components, which are written with WAMP client libraries, existing for multiple languages (currently 9). Every application component can be written in any of these, and it is possible to mix components written in multiple languages since all the interactions are via WAMP.

With Crossbar.io as a router, it is possible to create cross-platform applications. This enables application architectures such as Crossbar.io Node.

Crossbar.io is not just a WAMP router - it also provides and manages infrastructure for the application.

Features include:

- Integrated Static Web Server - it can serve HTML5 frontends directly.
- Component Hosting - start application components in any language together with Crossbar.io and it can manage their lifecycle
- Authentication and Authorization are configurable in Crossbar.io hosting WSGI applications
- HTTP Push Bridge for integration with legacy applications. This will often make Crossbar.io all the infrastructure you need besides your database.

Crossbar.io is high-performant, scalable, robust and secure, and distributed as Open Source under the AGPL v3 license.

It is Python code and runs on *nix, Windows and Mac OSX.

WAMP Protocol

WAMP provides Unified Application Routing in an open WebSocket protocol that works with different languages.

WAMP allows to build distributed systems out of application components which are loosely coupled and communicate in (soft) real-time.

The WAMP protocol offers two messaging patterns to allow components communicate:

- Routed Remote Procedure Calls (RPCs) - components register procedures and any other component can call this via Crossbar.io, with Crossbar.io handling the registrations, call and result routing.

- Publish & Subscribe (PubSub) - components subscribe to topics and publish to these, with Crossbar.io handling the subscriptions and dispatching.

The developers of WAMP think that applications have often a natural need for both forms of communication and it shouldn’t be required to use different protocols/means for those. Which is why WAMP provides both.

WAMP is easy to use, simple to implement and based on modern Web standards: WebSocket, JSON and URIs.

WAMP provides a feature called Unified Application Routing for applications: routing of both events (for PubSub) and routing of calls (for RPC) between applications components in one protocol.

Unified Routing is probably better explained by contrasting it with legacy approaches. Lets take the old "client-server" world. In the client-server model, a remote procedure call goes directly from the Caller to the Callee.

Remote procedure calls in the Client-Server model

In the client-server model, a Caller needs to have knowledge about where the Callee resides and how to reach it. This introduces a strong coupling between Caller and Callee which is bad, because applications can quickly become complex and unmaintainable. It is easy to explain how WAMP fixes that in a minute.

The problems coming from strong coupling between application components were long recognized and this (besides other requirements) lead to the publish-subscribe model.

In the publish-subscribe model a Publisher submits information to an abstract "topic", and Subscribers only receive information indirectly by announcing their interest on a respective "topic". Both do not know about each other. They are decoupled via the "topic" and via an intermediary usually called Broker.

A Broker decouples Publishers and Subscribers

A Broker keeps a book of subscriptions: who is currently subscribed on which topic. When a Publisher publishes some information ("event") to a topic, the Broker will look up who is currently subscribed on that topic: determine the set of Subscribers on the topic published to. And then forward the information ("event") to all those Subscribers.

The act of determining receivers of information (independently of the information submitted) and forwarding the information to receivers is called routing.

Now, WAMP translates the benefits of loose coupling to RPC. Different from the client-server model, WAMP also decouples Callers and Callees by introducing an intermediary - the Dealer.

Remote procedure calls in the Dealer model

Similar to a Broker's role with PubSub, the Dealer is responsible for routing a call originating from the Caller to the Callee and route back results or errors vice-versa. Both do not know about each other: where the peer resides and how to reach it. This knowledge is encapsulated in the Dealer.

With WAMP, a Callee registers a procedure at a Dealer under an abstract name: an URI identifying the procedure. When a Caller wants to call a remote procedure, it talks to the Dealer and only provides the URI of the procedure to be called plus any call arguments. The Dealer will look up the procedure to be invoked in his book of registered procedures. The information from the book includes where the Callee implementing the procedure resides, and how to reach it.

In effect, Callers and Callees are decoupled, and applications can use RPC and still benefit from loose coupling.

What if we combine both? Routed RPC and PubSub? When we combine a Broker and a Dealer we get what WAMP calls a Router.

A Router combines a Broker and a Dealer

A Router is capable of routing both calls and events, and hence can support flexible, decoupled architectures that use both RPC and PubSub.

Here is an example. Imagine having a small embedded device, like an Arduino Yun, with some sensors (e.g. temperature sensor) and actuators (e.g. a light or a motor) connected. Your aim is to integrate the device into an overall system where users are facing a frontend component to control the actuators while sensor values are continuously processed into a backend component.

By using WAMP, you can have a browser-based UI, the embedded device and your backend talk to each other in real-time.

Switching on a light on the device from the browser-based UI is naturally done by calling a remote procedure on the device. And the sensor values generated by the device continuously are naturally transmitted to the backend component (and possibly others) via publish & subscribe.

A.0.4 WebSocket

WebSocket is a protocol providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C. WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. The WebSocket Protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request. The WebSocket protocol makes more interaction between a browser and a website possible, facilitating live content and the creation of real-time application. This is made possible by providing a standardized way for the server to send content to the browser without being solicited by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way a two-way (bi-directional) ongoing conversation can take place between a browser and the server. The WebSocket protocol is currently supported in most major browsers including Google Chrome, Internet Explorer, Firefox, Safari and Opera.

A.0.5 AngularJS

AngularJS (commonly referred to as "Angular" or "Angular.js") is an open-source web application framework mainly maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. Angular.js purpose is simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures. The AngularJS library works by first reading the HTML page, which has embedded into it additional custom tag attributes. Angular interprets those attributes as directives to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code, or retrieved from static or dynamic JSON resources.

Appendix B

Relevant Source Code

Database Tables

Listing B.1 Current Daemon State options Table

```
CREATE TABLE CURRENT_STATE_OPTIONS (
valueID TEXT PRIMARY KEY NOT NULL,
value   TEXT             NOT NULL,
state   TEXT         KEY NOT NULL
);
```

Listing B.2 Current Daemon State

```
CREATE TABLE CURRENT_STATE (
id    INTEGER PRIMARY KEY    NOT NULL,
state TEXT            KEY    NOT NULL
);
```

Listing B.3 Escalation Table

```
CREATE TABLE ESCALATION (
```

```
id                   INTEGER PRIMARY KEY NOT NULL,
user_id              INTEGER                     ,
request_time         INTEGER             NOT NULL,
priviledge_grant_time INTEGER            NOT NULL,
force_enabling_time  INTEGER             NOT NULL
);
```

Listing B.4 Released Priviledge Table

```
CREATE TABLE PRIVILEDGE (
id                   INTEGER PRIMARY KEY NOT NULL,
user_id              INTEGER,
priviledge_expiration INTEGER,
no_answer_count      INTEGER
);
```

Listing B.5 Users table

```
CREATE TABLE USERS (
id                INTEGER PRIMARY KEY NOT NULL,
name              TEXT                NOT NULL,
username          TEXT                NOT NULL,
password          TEXT                NOT NULL,
user_role         INTEGER             NOT NULL,
user_can_escalate BOOLEAN             NOT NULL,
can_login         BOOLEAN             NOT NULL
);
```

Listing B.6 Roles Table

```
CREATE TABLE ROLES (
id                    INTEGER PRIMARY KEY  NOT NULL,
role_name             TEXT                 NOT NULL,
role_can_escalate     BOOLEAN              NOT NULL,
priviledge_will_expire BOOLEAN             NOT NULL,
is_super_user         BOOLEAN              NOT NULL,
can_edit_user         BOOLEAN              NOT NULL,
session_will_expire   BOOLEAN              NOT NULL
);
```

Listing B.7 Sessions Table

```
CREATE TABLE SESSIONS (
user_id INTEGER PRIMARY KEY NOT NULL,
token   TEXT                NOT NULL,
expire  INTEGER             NOT NULL
);
```

Listing B.8 Audit Table

```
CREATE TABLE AUDITLOG (
id          INTEGER PRIMARY KEY  NOT NULL,
user_id     INTEGER              NOT NULL,
timestamp   INTEGER              NOT NULL,
description TEXT                 NOT NULL
);
```

Appendix C

Example of Datacard

Listing C.1 Audit Table

```
{
  "DETECTOR_GEOMETRY": {
    "PMTS": "6",
    "FLOORS": "14",
    "TOWERS": "4"
  },
  "INTERNAL_SW_PARAMETERS": {
    "DELTA_TS": "200",
    "PMT_BUFFER_SIZE": "1000000",
    "STS_READY_TIMEOUT": "5",
    "TTS_READY_TIMEOUT": "30",
    "STS_IN_MEMORY": "100"
  },
  "FCM_ENDPOINTS": [
    {
```

```
      "DATA_HOST": "172.16.1.101",
      "DATA_PORT": "16000"
    },
    {
      "DATA_HOST": "172.16.1.101",
      "DATA_PORT": "16001"
    },
    {
      "DATA_HOST": "172.16.1.101",
      "DATA_PORT": "16002"
    },
    {
      "DATA_HOST": "172.16.1.101",
      "DATA_PORT": "16003"
    },
    {
      "DATA_HOST": "172.16.1.101",
      "DATA_PORT": "16004"
    },
    {
      "DATA_HOST": "172.16.1.101",
      "DATA_PORT": "16005"
    },
    [CUT..]

  ],
```

```
  "HM": {
    "LOG_LEVEL": "DEBUG",
    "BASE_CTRL_PORT": "16100",
    "DUMP_FLAG": "0",
    "DUMP_FILENAME_PREFIX": "\/tmp\/hm_dump_",
    "DUMP_MAX_SIZE": "500",
    "HOSTS": [
      {
        "CTRL_HOST": "192.168.253.114",
        "N_INSTANCES": "1"
      },
      {
        "CTRL_HOST": "192.168.253.115",
        "N_INSTANCES": "1"
      },
      {
        "CTRL_HOST": "192.168.253.116",
        "N_INSTANCES": "1"
      },
      {
        "CTRL_HOST": "192.168.253.117",
        "N_INSTANCES": "1"
      }
    ]
  },
  "TCPU": {
```

```
68    "LOG_LEVEL": "DEBUG",
69    "DUMP_FLAG": "0",
70    "DUMP_FILENAME_PREFIX": "\/tmp\/tcpu_dump_",
71    "DUMP_MAX_SIZE": "500",
72    "BASE_CTRL_PORT": "16200",
73    "BASE_DATA_PORT": "16300",
74    "OFFLINE_FLAG": "0",
75    "SIMULATION_FILENAME": "\/tmp\/simulated_events.txt",
76    "PARALLEL_TTS": "2",
77    "PLUGINS_DIR": "\/tmp\/plugins",
78    "HOSTS": [
79      {
80        "CTRL_HOST": "192.168.253.118",
81        "DATA_HOST": "10.0.80.118",
82        "N_INSTANCES": "1"
83      },
84      [CUT]
85    ],
86       "TRIGGER_PARAMETERS": {
87            "L1_EVENT_WINDOW_HALF_SIZE": "600",
88            "L1_DELTA_TIME_SC": "4",
89            "L1_DELTA_TIME_FC": "20",
90            "L1_CHARGE_THRESHOLD": "500",
91            "L1_FLAG_RT": "1",
92            "L1_FREQUENCY_RT": "5",
93            "L1_DELTA_TIME_RT": "200000",
```

```
            "L1_DELTA_TIME_SEQHIT": "200",
            "L1_N_SEQHIT": "7"
        },
        "PLUGINS": {
            "RANDOM": {
                "NAME": "TrigRandom",
                "ID": "0",
                "PARAMETERS": {}
            },
            "SCALER_10": {
                "NAME": "TrigScaler",
                "ID": "1",
                "PARAMETERS": {
                    "SCALE_FACTOR": "10"
                }
            }
        },
    "TSV": {
      "LOG_LEVEL": "DEBUG",
      "CTRL_HOST": "192.168.253.113"
    },
    "EM": {
      "CTRL_HOST": "192.168.253.112",
      "DATA_HOST": "192.168.253.112",
      "DATA_PORT": "16400",
      "NETWORK_THREADS": "1",
```

```
120     "LOG_LEVEL": "DEBUG",
121     "LOG_TO_SYSLOG": "0",
122     "FILE_MAX_SIZE": "2000000000",
123     "PT_FILE_PREFIX": "/home/tridas/nemo_f3_pt",
124     "PT_FILE_POSTFIX": ".dat"
125   },
126   "MONITOR": {
127     "CTRL_HOST": "lxantares3.bo.infn.it",
128     "CTRL_PORT": "9999",
129     "TIME_INTERVAL": "3"
130   },
131   "TSC": {
132     "DATACARD_SHAREDDIR": "/lxstorage1_home/km3/datacard/tsc"
133   },
134   "TOWER_0": {
135     "FLOOR_1": {
136       "PMT_0": {
137         "X": "38.325",
138         "Y": "3.122",
139         "Z": "130.297",
140         "CX": "-0.884",
141         "CY": "-0.467",
142         "CZ": "0",
143         "TIME_OFFSET": "5",
144         "PEDESTAL": "5",
145         "THRESHOLD": "25"
```

```
    },
    "PMT_1": {
      "X": "38.767",
      "Y": "3.356",
      "Z": "130.017",
      "CX": "0",
      "CY": "0",
      "CZ": "-1",
      "TIME_OFFSET": "5",
      "PEDESTAL": "5",
      "THRESHOLD": "25"
    },
    "PMT_2": {
      "X": "41.64",
      "Y": "4.875",
      "Z": "130.297",
      "CX": "-0.331",
      "CY": "0.625",
      "CZ": "-0.707",
      "TIME_OFFSET": "5",
      "PEDESTAL": "5",
      "THRESHOLD": "25"
    },
    "PMT_3": {
      "X": "42.082",
      "Y": "5.108",
```

```
      "Z": "130.297",
      "CX": "0.331",
      "CY": "-0.625",
      "CZ": "-0.707",
      "TIME_OFFSET": "5",
      "PEDESTAL": "5",
      "THRESHOLD": "25"
    },
    "PMT_4": {
      "X": "44.955",
      "Y": "6.628",
      "Z": "130.017",
      "CX": "0",
      "CY": "0",
      "CZ": "-1",
      "TIME_OFFSET": "5",
      "PEDESTAL": "5",
      "THRESHOLD": "25"
    },
    "PMT_5": {
      "X": "45.397",
      "Y": "6.861",
      "Z": "130.297",
      "CX": "0.884",
      "CY": "0.467",
      "CZ": "0",
```

```
        "TIME_OFFSET": "5",
        "PEDESTAL": "5",
        "THRESHOLD": "25"
      }
    },
    "FLOOR_2": {
      "PMT_0": {
        "X": "43.731",
        "Y": "1.455",
        "Z": "110.297",
        "CX": "0.467",
        "CY": "-0.884",
        "CZ": "0",
        "TIME_OFFSET": "5",
        "PEDESTAL": "5",
        "THRESHOLD": "25"
      },
      "PMT_1": {
        "X": "43.497",
        "Y": "1.897",
        "Z": "110.017",
        "CX": "0",
        "CY": "0",
        "CZ": "-1",
        "TIME_OFFSET": "5",
        "PEDESTAL": "5",
```

```
        "THRESHOLD": "25"
      },
      "PMT_2": {
        "X": "41.978",
        "Y": "4.771",
        "Z": "110.297",
        "CX": "0.625",
        "CY": "0.331",
        "CZ": "-0.707",
        "TIME_OFFSET": "5",
        "PEDESTAL": "5",
        "THRESHOLD": "25"
      },
      "PMT_3": {
        "X": "41.744",
        "Y": "5.213",
        "Z": "110.297",
        "CX": "-0.625",
        "CY": "-0.331",
        "CZ": "-0.707",
        "TIME_OFFSET": "5",
        "PEDESTAL": "5",
        "THRESHOLD": "25"
      },
      [CUT...]
  }
```

```
250 }
```