Università degli Studi di Pisa

Facoltà di Scienze Matematiche
Fisiche e Naturali

Laurea Specialistica in Tecnologie Informatiche

Sviluppo e supporto al Data Quality Monitoring dell'esperimento CMS

Tesi di Laurea

Andrea Carboni

PISA 10 Ottobre 2008

Relatori:
Prof. Marco Danelutto (Università degli Studi di Pisa)
Dott. Emilio Meschi (CERN-Ginevra)

Anno Accademico 2007-2008

Alla mia famiglia

Indice

Sommario

In questa tesi viene descritto il lavoro di studio ed implementazione che ho svolto in riferimento al *Data Quality Monitoring*, framework che si occupa di rendere disponibile al personale al lavoro sull'esperimento CMS, presso il CERN di Ginevra, un sistema di monitoring omogeneo per quanto concerne gran parte delle attività di acquisizione dati da esso previste.

Il primo capitolo di questa tesi contiene un'introduzione al laboratorio del CERN, all'acceleratore LHC ed inquadra in tale contesto l'esperimento CMS, facendo particolare riferimento ad aspetti legati alle risorse computazionali richieste.

Il secondo capitolo presenta una descrizione dettagliata del framework CMSSW andando a toccare nella parte conclusiva la parte relativa alla visualizzazione. Quest'ultima, come sarà esposto nei capitoli successivi, è stata fra le problematiche che mi hanno interessato direttamente.

A partire dal terzo capitolo inizia la descrizione del lavoro da me effettivamente svolto. Poichè tale lavoro ha interessato l'intero framework DQM, dalle comunicazioni fra moduli fino all'interazione grafica con l'utente, il capitolo offre una panoramica in ordine temporale di tutte le maggiori modifiche da me apportate o alle quali ho contribuito attivamente, sia per quanto riguarda l'implementazione che per quanto riguarda la fase di design. La prima parte descrive un'importante modifica architetturale del framework, mentre la seconda descrive l'evoluzione dell'interfaccia grafica,

evidenziando problematiche e soluzioni che saranno viste in dettaglio nei capitolo successivi.

Il quarto capitolo descrive la parte di implementazione, riferita alle tematiche viste nel precedente capitolo, descrivendo soluzioni da me implementate, che sono state effettivamente adottate dal framework e che hanno fatto parte integrante di tutta l'evoluzione già descritta.

Il quinto e conclusivo capitolo, vede il design di due diverse soluzioni riguardante alcune problematiche viste in precedenza, la seconda delle quali ha portato all'implementazione, avvenuta nei mesi successivi alla fine del mio lavoro, dell'interfaccia web descritta nel terzo capitolo.

Capitolo 1

Introduzione

1.1 La fisica delle particelle elementari

La ricerca sulla particelle elementari cerca di dare risposta a due domande basilari: quali sono i costituenti elementari della materia e quali sono le forze che ne governano il comportamento a livello elementare. Per spezzare la materia nei suoi costituenti si fa ricorso a collisioni fra particelle di materia (ad esempio protoni o elettroni) accelerate in fasci fino ad altissime energie. I protoni, ad esempio, possono acquistare in un acceleratore un'energia cinetica fino a migliaia di volte la loro energia di riposo [1]. Questa energia viene restituita nella collisione dando luogo alla frammentazione dei protoni che si scontrano e alla creazione di nuove particelle, le più interessanti di durata effimera ma di grande massa. Lo studio di questi eventi di collisione rappresenta il punto centrale della sperimentazione sulle particelle elementari.

[1] In base alla teoria della relatività la massa è una forma di energia e l'una può trasformarsi nell'altra. A una particella di massa m è associata l'energia mc^2, detta energia di riposo. Per creare una particella di massa m occorre spendere questa quantità di energia; la stessa energia si rende disponibile nella distruzione della particella.

1.1.1 Il laboratorio del CERN

Il CERN [1], Organizzazione Europea per la Ricerca Nucleare, con sede a Ginevra, rappresenta a livello mondiale il più grande centro di ricerca sperimentale sulla fisica delle particelle elementari. Qui i fisici vengono per indagare sulla composizione della materia e sulle forze che la governano e la tengono unita. Il CERN esiste principalmente per fornire loro gli strumenti necessari per progredire nella ricerca: tali strumenti sono gli acceleratori, che accelerano le particelle quasi fino alla velocità della luce. Attorno agli acceleratori sono montati gli esperimenti con i loro rivelatori, che rendono misurabili (e quindi in un certo senso "visibili") le collisioni delle particelle accelerate.

Figura 1.1: I 20 stati che fanno parte del CERN

Fondato nel 1954, il laboratorio è stato una delle prime collaborazioni europee ed include ad oggi venti stati membri (Figura 1.1), anche se molte altre sono le nazioni non europee che in modi diversi sono coinvolte nei progetti di ricerca che vengono portati avanti dal CERN.

Gli stati membri hanno doveri e privilegi speciali: contribuiscono ai costi necessari per portare avanti i programmi del CERN, includendo la costruzione degli acceleratori e degli apparati rivelatori. Sono rappresentati nel *Consiglio*, organo responsabile per quanto concerne tutte le più importanti decisioni sull'organizzazione

e la politica scientifica e tecnica del laboratorio. Gli istituti di ricerca e le università dei paesi membri possono proporre delle ricerche sperimentali che vengono vagliate da appositi comitati scientifici ed eventualmente approvate. In generale, data la complessità ed i costi, gli esperimenti sono proposti da *Collaborazioni* internazionali. In caso di approvazione, gli esperimenti hanno diritto all'utilizzo gratuito delle macchine acceleratrici. Per contro, il CERN non è un ente finanziatore della ricerca, i rivelatori sono pagati e costruiti dagli istituti proponenti l'esperimento. Il CERN può tuttavia anche rivestire il ruolo di istituto di ricerca alla stregua degli istituti dei paesi membri, partecipando ad una Collaborazione, ed in tal caso paga e costruisce la sua parte di esperimento.

1.1.2 L'acceleratore LHC

Il complesso di acceleratori del CERN è una successione di macchine che, accelerando l'una dopo l'altra il flusso di particelle, portano quest'ultimo a valori energetici sempre più alti. Quando l'energia finale in gioco è molto elevata non è infatti possibile realizzare l'accelerazione in un'unica macchina e se ne dispongono diverse in cascata. Il fiore all'occhiello del complesso è rappresentato dal Large Hadron Collider (LHC), attualmente in fase di attivazione. Si tratta di un acceleratore circolare a due fasci di protoni circolanti in senso opposto e della medesima energia, ossia 7 TeV per fascio [2].

Questo tipo di macchina si dice in inglese *collider* ed ha il vantaggio che tutta l'energia cinetica a disposizione (7+7 = 14 TeV) è disponibile per creare nuove particelle [3].

[2] 1 TeV = 10^{12} eV, corrisponde a $\approx 10^9$ l'energia a riposo del protone. Con 1 TeV si potrebbero creare 10^9 protoni, ossia circa 1.6 10^{-27} kg di materia.

[3] In realtà , a causa della natura composita del protone, l'energia disponibile per creare nuove particelle è solo una frazione dell'energia totale nel centro di massa, e dipende dalla frazione di impulso portata dai singoli componenti del protone che effettivamento si urtano.

LHC ha una circonferenza di ben 27 km ed è completamente sotterraneo. Una piccola parte si trova in Svizzera ma il grosso si trova in territorio francese. Il tunnel sotterraneo ospitava fino all'anno 2000 il collider elettroni-positroni LEP, che dopo dieci anni di servizio è stato smantellato.

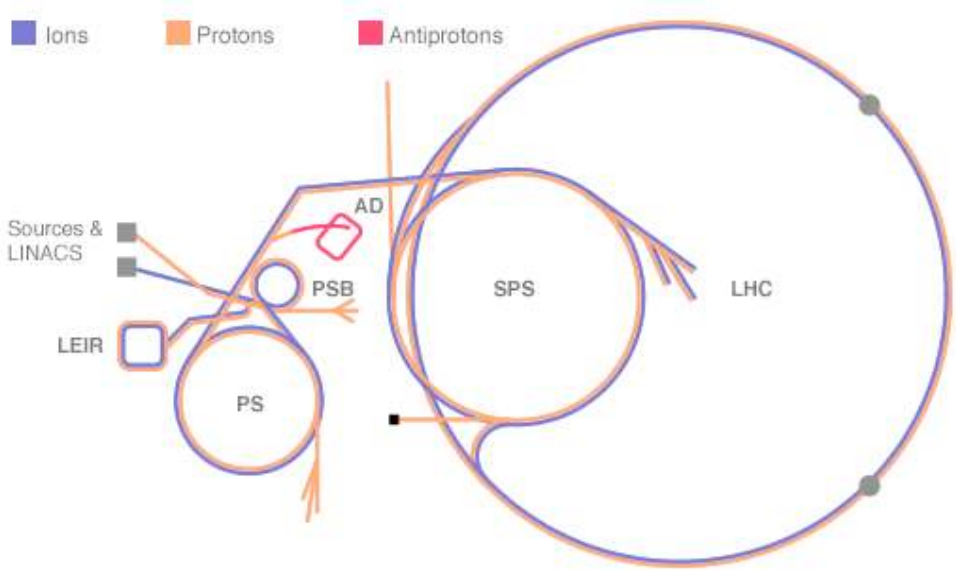

Figura 1.2: Il complesso di acceleratori del CERN.

1.1.3 Gli esperimenti a LHC

Un esperimento nasce sempre da un'idea, una teoria formulata che necessita di essere confermata oppure confutata.

Tre sono gli aspetti che caratterizzano un esperimento al CERN:

- la tematica scientifica (ciò che si vuole scoprire)
- l'apparato sperimentale (il complesso di rivelatori usato)
- la Collaborazione (il gruppo internazionale di lavoro)

Dopo che il fascio di particelle energetiche è stato prodotto all'interno di un acceleratore, due sono le possibili strade percorribili: o viene fatto collidere con un bersaglio fisso, oppure con un altro fascio simile e proveniente da direzione opposta.

In LHC si segue la seconda via e per questo motivo la macchina si chiama *collider* o, in italiano, *collisore*. In questo modo tutta l'energia dei due fasci (14 TeV) viene utilizzata nella collisione. Il risultato della collisione è la produzione di un elevato numero di nuove particelle, il comportamento delle quali viene monitorato e studiato attraverso l'uso dei rivelatori.

Su LHC sono installati i seguenti quattro esperimenti: ALICE (*A Large Ion Collider Experiment*), ATLAS (*A Toroidal LHC ApparatuS*), CMS (*Compact Muon Solenoid*) ed LHCb (*Large Hadron Collider beauty*).

Mentre ALICE e LHCb perseguono delle linee scientifiche mirate allo studio di particolari fenomeni, ATLAS e CMS sono progettati per uno studio generale delle collisioni che si svolgono a queste elevatissime energie (esperimenti *general-purpose*). In questi urti si ricreano a livello elementare le stesse energie del cosiddetto *Big Bang* dell'universo. In particolare la fisica teorica prevede in questa situazione la produzione della cosiddetta particella di *Higgs*. Scopo principale degli esperimenti ATLAS e CMS è la verifica appunto della sua esistenza.

Il gruppo di sviluppo di cui ho fatto parte durante il mio periodo di tesi lavora per l'esperimento di alta energia CMS, in particolare allo sviluppo e mantenimento del *Data Quality Monitoring*, framework che ha come obiettivo primario quello di garantire la qualità dei dati raccolti durante la fase di raccolta e selezione dati.

1.2 L'esperimento CMS

L'esperimento CMS è, insieme ad ATLAS, uno dei due esperimenti *general purpose* approvati per LHC, alla costruzione del suo rivelatore hanno lavorato circa 2600 persone provenienti dai 180 diversi istituti scientifici della Collaborazione, di cui 16 italiani. Il rivelatore ha la forma di un cilindro lungo 21 metri, di diametro 16 metri e

pesante approssimativamente 12500 tonnellate (Figura 1.3), esso è installato in una caverna sotterranea posta lungo l'anello principale di LHC a 100 metri di profondità presso Cessy, in Francia, appena fuori dai confini di Ginevra.

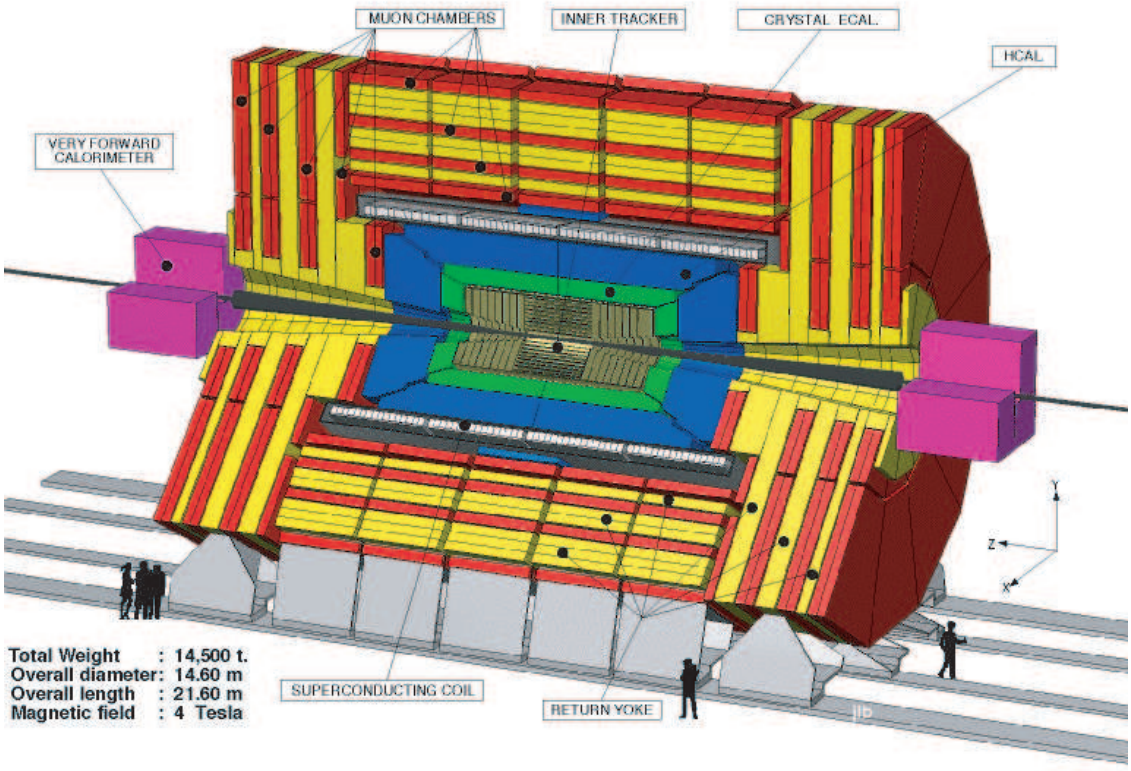

Figura 1.3: Spaccato del rivelatore dell'esperimento CMS.

I principali obiettivi dell'esperimento sono:

- esplorare la fisica alla scala energetica dei TeV;
- verificare l'esistenza del *bosone di Higgs*;
- cercare prove che dimostrino che è possibile andare oltre il cosiddetto *Modello Standard* della fisica delle particelle, e stabilire la validità o meno di teorie quali la *Supersimmetria*, le stringhe o le *Extra dimensioni*;
- studiare fenomeni legati a principi di simmetria (violazione di *CP*).

La maggior parte dei rivelatori per la fisica delle particelle elementari sono realizzati attorno ad un sistema magnetico di qualche tipo allo scopo di agevolare

la misura dell'impulso delle particelle cariche. CMS non si allontana da questo modello: esso utilizza un grosso solenoide superconduttore, lungo circa 12 metri e con un diametro interno di approssimativamente 6 metri per produrre un campo magnetico di 4 Tesla, circa centomila volte quella del campo magnetico terrestre. Questo solenoide è il più grande magnete di questo tipo mai costruito, al suo interno contiene tutti i dispositivi di tracciamento e di calorimetria, risultando così in un apparato relativamente compatto.

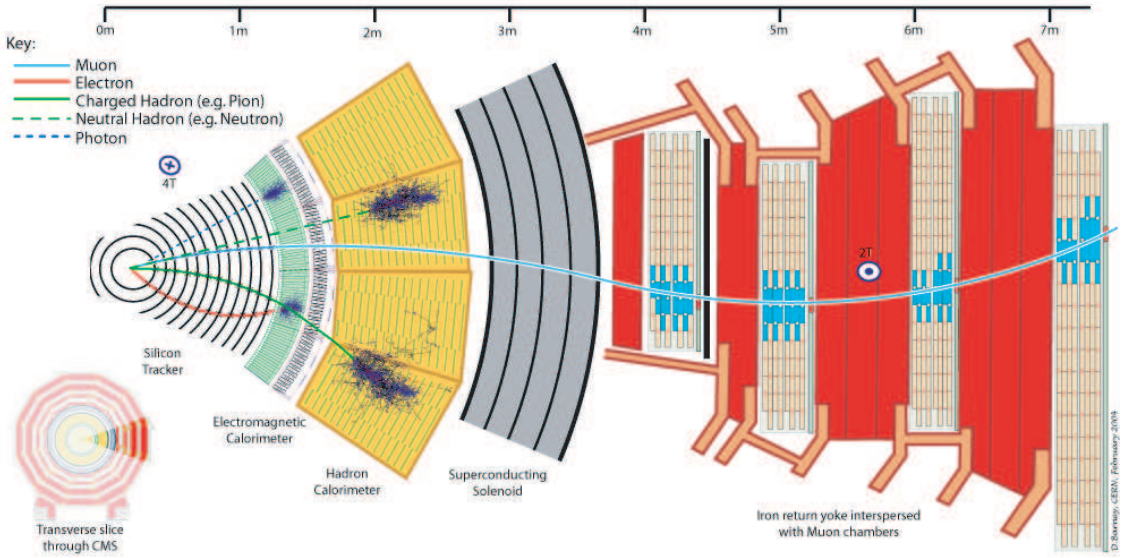

Figura 1.4: Sezione del rivelatore CMS con i vari strati e i sottorivelatori che misurano le caratteristiche e le traiettorie delle particelle prodotte nell'urto. Il campo magnetico (perpendicolare al foglio) deflette le particelle elettricamente cariche e permette di misurarne la quantità di moto (impulso).

Analizzando una sezione del rivelatore (Figura 1.4) è possibile individuare diversi strati (*layer*), ciascuno dei quali responsabile di ben determinate funzioni nel processo di rilevazione delle collisioni. Il centro del rivelatore rappresenta la regione di collisione, la zona in cui i fasci di protoni collidono l'uno con l'altro all'interno della camera a vuoto di LHC. Tali fasci sono organizzati in *pacchetti* di protoni, ciascuno dei quali ne contiene approssimativamente 10^{11}. Le particelle sono però talmente "piccole" che le probabilità di una collisione sono bassissime: infatti,

quando i pacchetti di due fasci si incrociano, su $2 \cdot 10^{11}$ protoni le collisioni sono solo una ventina.

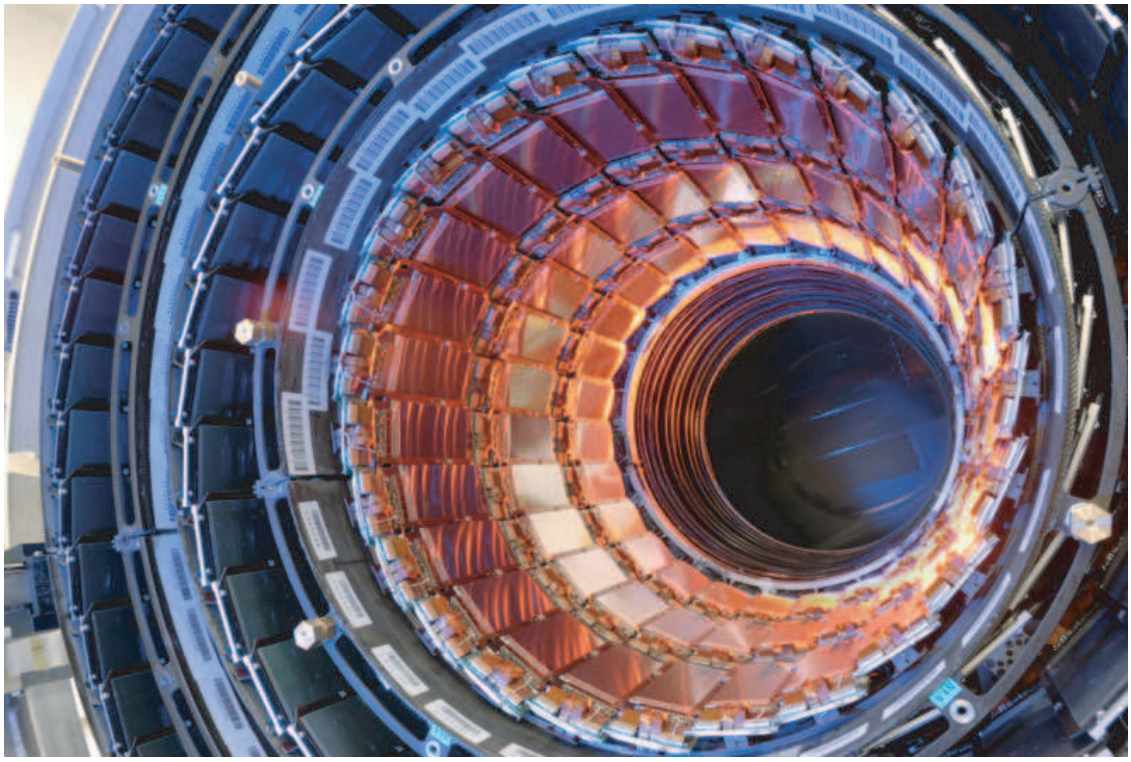

Figura 1.5: Immagine dell'interno del tracciatore prima dell'inserimento nel rivelatore.

Quando due protoni collidono a livelli di energia così elevati, lo scambio di massa ed energia genera processi che possono portare alla creazione di particelle non riscontrabili nel mondo comune. La maggior parte di questi processi sono già ben noti, solitamente su un miliardo di collisioni solo cento di esse rappresentano eventi interessanti. I pacchetti sono separati fra loro temporalmente di appena 25 ns, in modo da avere 40 milioni di collisioni al secondo. La zona di collisione è circondata da un dispositivo chiamato *tracciatore* (in inglese *tracker*) (Figura. 1.5), situato nello strato più interno del rivelatore.

Esso, come si evince dal nome, è in grado di riconoscere le tracce lasciate dalle particelle cariche generate in seguito a una collisione e di misurare il loro impulso.

Il tracciatore è formato da sensori di silicio disposti in 13 cilindri concentrici all'asse dei fasci che coprono un'area complessiva pari a quella di un campo da tennis, con ben 75 milioni di canali di lettura per determinare la posizione delle particelle che li attraversano (Figura. 1.6).

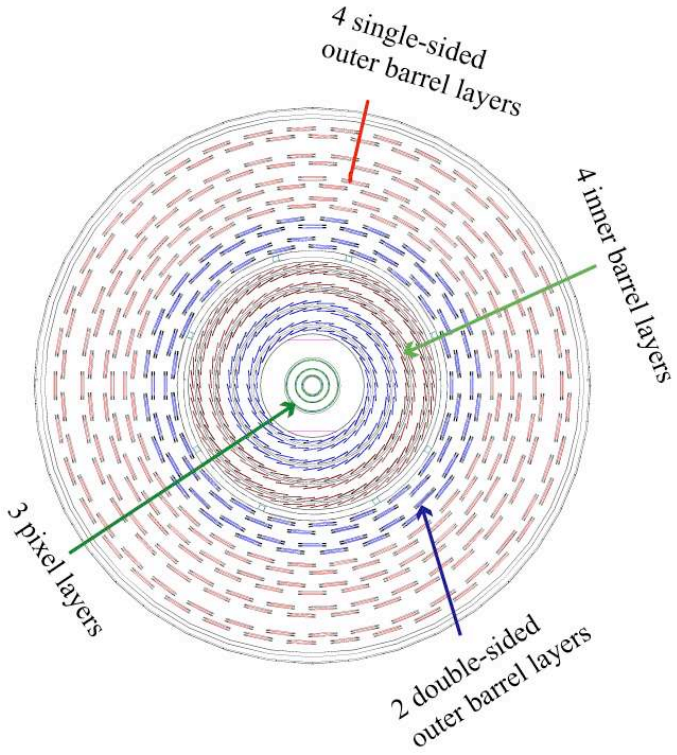

Figura 1.6: Schema della parte centrale del tracciatore con i vari piani di rivelatori al silicio. Il raggio della parte più interna è di 25 cm, quello esterno di 1 metro.

All'esterno del tracciatore il secondo e il terzo strato sono occupati rispettivamente dal calorimetro elettromagnetico e dal calorimetro adronico. Il calorimetro elettromagnetico permette di misurare con precisione le energie di elettroni e fotoni mentre il calorimetro adronico quelle degli adroni, ossia particelle quali protoni, neutroni, pioni e kaoni. Proseguendo verso l'esterno, il quarto strato è costituito dal solenoide citato in precedenza, mentre il quinto e ultimo strato è costituito dal *Muon Detector*, che permette di ricostruire le tracce lasciate da particelle chiamate muoni, ed è in realtà costituito a sua volta da tre componenti: *Drift Tubes* (DT), *Catho-*

de Strip Chambers (CSC) e *Resistive Plate Chambers* (RPC). Complessivamemte il rivelatore comprende circa 100 milioni di canali di lettura.

Attualmente il rivelatore CMS è in fase di test che utilizzano i raggi cosmici (che possono penetrare lo strato di roccia sovrastante l'esperimento) e i primi fasci di particelle immessi nell'acceleratore LHC. La Figura 1.7 mostra un evento di raggio cosmico che ha attraversato il rivelatore ed è stato ricostruito online.

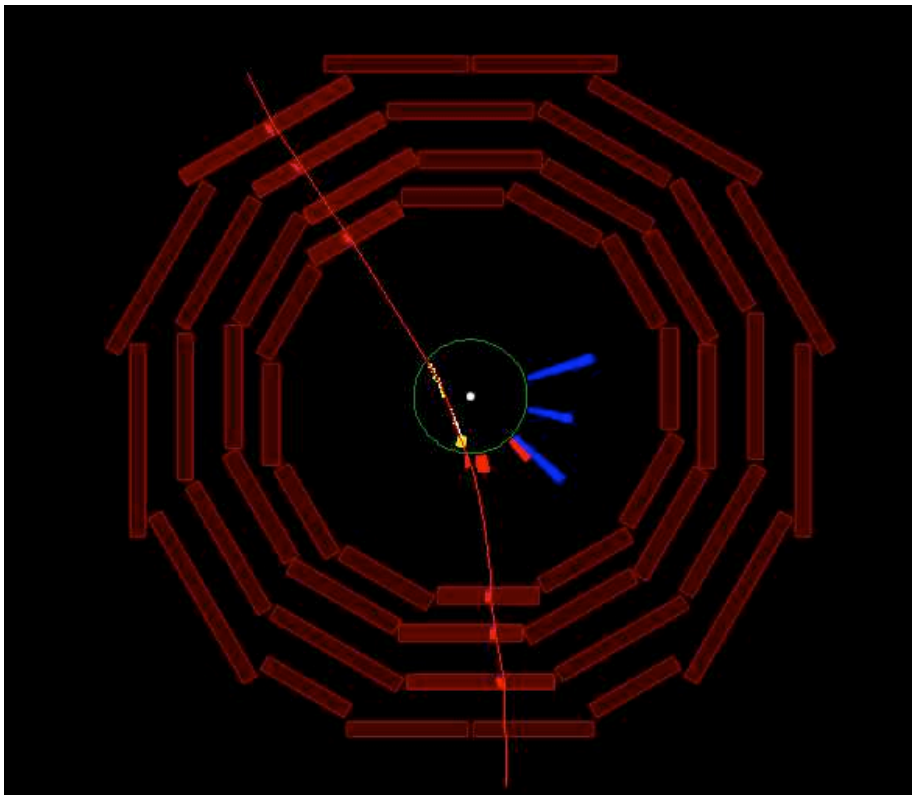

Figura 1.7: Traccia di raggio cosmico rivelata e ricostruita nei vari strati dell'apparato CMS. La traiettoria è stata curvata dal campo magnetico del solenoide (vedi testo).

La Figura 1.8 invece mostra un evento raffigurante la produzione di una particella supersimmetrica.

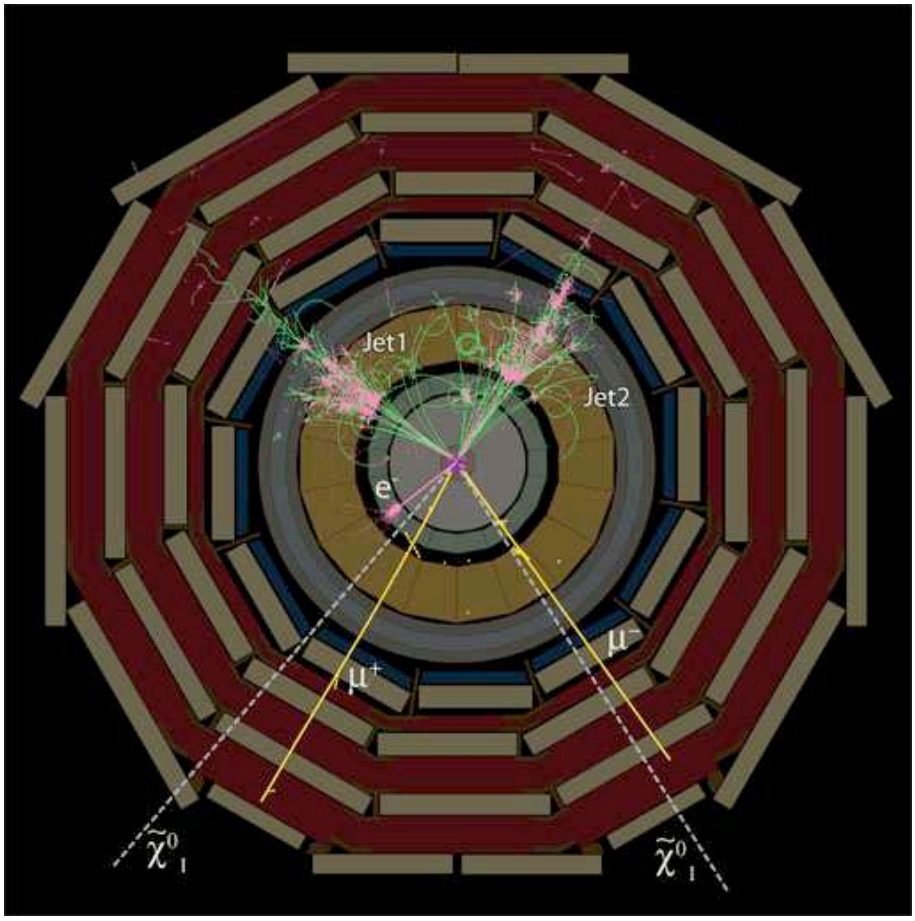

Figura 1.8: Simulazione di produzione e decadimento di una particella supersimmetrica. Tutti gli strati del rivelatore CMS sono necessari per ricostruire l'evento.

1.2.1 Il modello computazionale di CMS

CMS rappresenta una sfida non soltanto dal punto di vista delle scoperte fisiche o da quello ingegneristico, per la costruzione e attivazione di sistemi estremamente complessi come i rivelatori, ma anche da quello del volume di dati e delle risorse computazionali necessarie, sia da un punto di vista di elaborazione che di memorizzazione. Tali requisiti infatti sono di almeno un ordine di grandezza superiori rispetto al passato. Come prima ricordato, il numero complessivo di canali di lettura dell'esperimento è di circa 100 milioni che vengono processati ogni 25 ns dal sistema di acquisizione. Una frazione dei 40 milioni di eventi al secondo, ritenuta interessante, viene ulteriormente processata e immagazzinata su disco. Ad esperimento in esecuzione verranno immagazzinati ogni anno 1 *Petabyte* [4] di dati relativi al rivelatore, tutto ciò per un periodo di tempo lungo almeno quindici anni.

Centri di calcolo CMS

Per motivazioni sia tecniche che economiche risulta impossibile soddisfare tutti i requisiti dell'esperimento in un unico sito, questo sia da un punto di vista computazionale che di immagazzinamento dati. Inoltre, la maggior parte dei collaboratori CMS, sebbene non abbiano base al CERN, hanno comunque accesso a significative risorse esterne che risultano di grande aiuto per portare avanti nel migliore dei modi la parte relativa al *computing* dell'esperimento.

Per le ragioni appena citate, l'ambiente computazionale dell'esperimento CMS è stato pensato e sviluppato come un sistema distribuito di servizi e risorse che interagiscono tra di loro come servizi GRID [2]. L'insieme dei servizi ed il loro comportamento includono le risorse computazionali, di stoccaggio e di connettività

[4] 1 *Petabyte* (PB) corrisponde a 10^{15} *byte* o 1000 *Terabyte* (TB)

che CMS usa per l'elaborazione dati, il loro immagazzinamento, il generatore di eventi MonteCarlo [3] e tutte le altre attività di computing ad esse correlate.

I centri computazionali a disposizione di CMS e sparsi per il pianeta sono distribuiti e configurati in un'architettura stratificata che funziona come un singolo sistema: a ciascuno dei tre strati (chiamati *Tier*) corrispondono differenti risorse e servizi (Figura. 1.9).

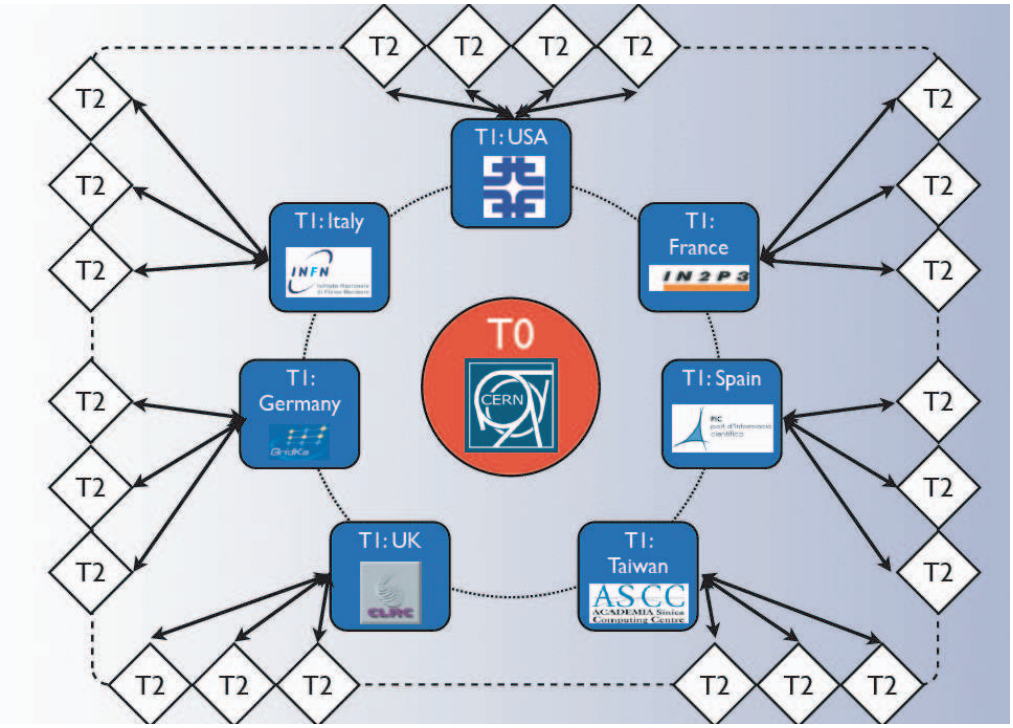

Figura 1.9: Immagine esemplificativa della struttura a Tier del GRID per LHC.

Tier-0 (T0)

Il primo Tier nel modello CMS, per il quale esiste un solo sito, il CERN stesso, è noto come Tier-0 (T0). Esso è responsabile di svariate funzioni, le principali sono:

- accettare dati RAW [5] provenienti dal *CMS Online Data Acquisition System (DAQ)*;

[5] Dati relativi all'evento completo come registrati dall'esperimento.

- rielaborare i dati ricevuti dal DAQ in *dataset* primari in funzione delle informazioni di *trigger* [6];
- archiviare su nastro i dataset;
- ridistribuire i dati appena archiviati verso i Tier-1 in modo che ogni dato RAW risulti salvato in due copie, una al CERN ed una su un Tier-1;
- eseguire le procedure di calibrazione *PromptCalibration* in modo da ottenere le costanti di calibrazione necessarie per eseguire gli algoritmi di ricostruzione;
- eseguire gli algoritmi di ricostruzione *PromptReconstruction*;
- distribuire i dati RECO [7] e AOD [8] a ciascun Tier-1.

Tier-1 (T1)

Allo stato attuale esiste un insieme di sette siti Tier-1 (T1). A ciascuno di essi corrisponde un grosso centro in uno dei paesi della Collaborazione CMS. I siti Tier-1 sono solitamente utilizzati per attività centralizzate su larga scala e possono fornire o ricevere dati da tutti i siti Tier-2. Ciascun centro T1:

- riceve un sottoinsieme dei circa cinquanta dataset da parte dei T0;
- provvede ad archiviare su nastro i dati RAW che riceve;
- fornisce potenza di calcolo per:
  - *re-reconstriction*;
  - *skimming*;
  - calibrazione;
  - estrazione AOD;
  - attività di analisi *data-intensive*;
- conserva un'intera copia dell'AOD;

[6] Nella fisica delle particelle un "trigger" è un sistema che utilizza semplici criteri per decidere quali eventi rivelati dal detector conservare, quando solo una piccola frazione di essi può essere registrata.

[7] Output del primo filtraggio all'interno del T0. Questo strato contiene i cosiddetti reconstructed physics object e rimane comunque molto dettagliato

[8] Versione 'distillata' delle informazioni RECO utilizzata nella maggior parte delle analisi.

- distribuisce RECO, skim e AOD al CERN, agli altri T1 ed ai T2 ad esso associati;
- fornisce stoccaggio sicuro e ridistribuzione degli eventi Monte Carlo generati dai T2.

Tier-2 (T2)

Essi rappresentano centri più piccoli rispetto ai T1 e sono solitamente localizzati all'interno di università. Sono caratterizzati da notevole disponibilità in termini di potenza di calcolo, che viene utilizzata per l'analisi dettagliata da parte di singoli fisici o gruppi, gli studi di calibrazione e la produzione di eventi simulati. Per contro i siti T2 non prevedono la possibilità di archiviazione su nastro ed hanno limitato spazio disco. In generale le funzionalità dei siti T2 si possono riassumere in:

- servizi per comunità locali;
- analisi GRID per l'intero esperimento;
- simulazione Monte Carlo per l'intero esperimento.

Il Tier-1 Italiano

Inaugurato il 17 Novembre 2005 e collocato presso il CNAF dell'INFN, negli edifici del Dipartimento di Fisica dell'Università di Bologna, questo sistema di calcolo ad alte capacità dell'istituto Nazionale di Fisica Nucleare (INFN) è un nodo di primo livello, al pari di soli altri dieci centri nel mondo, e parte essenziale dell'infrastruttura sviluppata attraverso progetti di Grid Computing (Grid.it), Europei (EGEE) ed Internazionali (LHC Computing Grid).

Il centro si propone a livello di eccellenza mondiale nel fornire supporto a tutte quelle comunità scientifiche che si basano sull'elaborazione intensiva e sull'archiviazione di grandi moli di dati. È l'unica sede italiana di primo livello (Tier-1)

che dovrà fornire le ingenti risorse di calcolo necessarie per elaborare, archiviare e gestire l'imponente quantità di dati che sarà prodotta dai quattro esperimenti del Large Hadron Collider e servirà in primo luogo più di 6000 fisici ditribuiti in tutto il mondo.

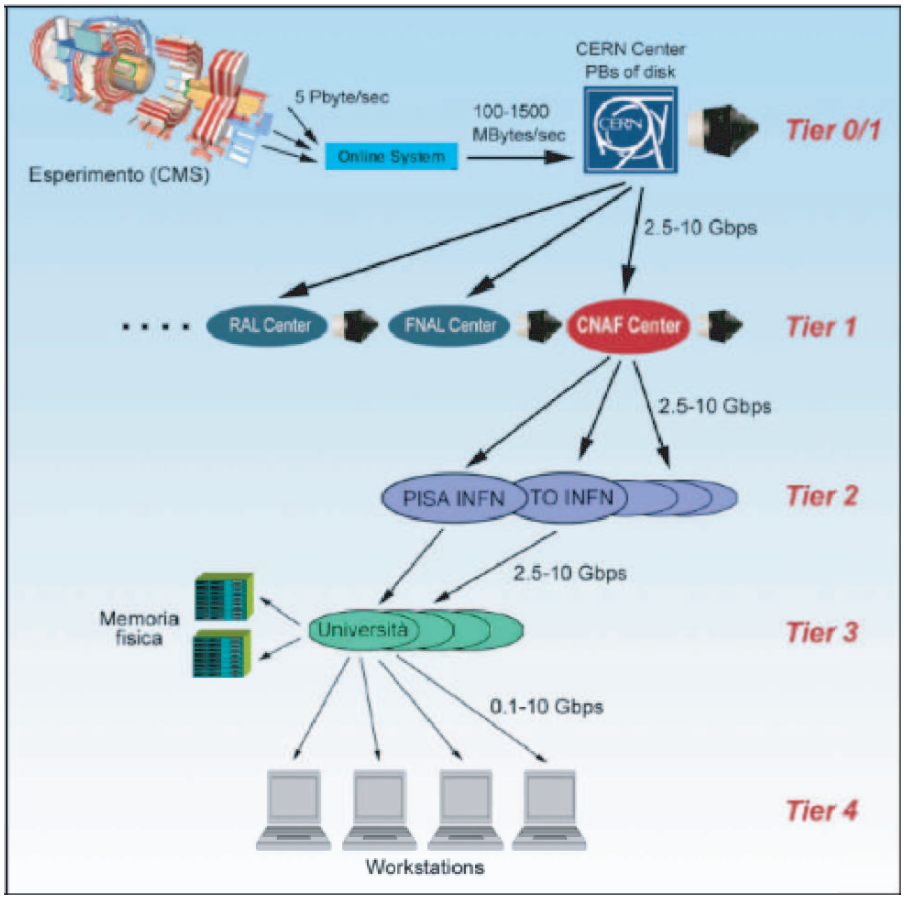

Figura 1.10: Struttura Tier-1 Italiana.

L'infrastruttura Grid è utile non solo alla Fisica, ma a quasi tutti i settori della Scienza e della Società moderna che ormai si caratterizzano per una enorme quantità di dati che sono difficilmente utilizzabili in modo diretto dagli utenti. Così avviene, ad esempio, per i dati prodotti dai satelliti, dai sensori ambientali, dalle TAC e Risonanze Magnetiche, dalle transazioni di Borse e della Finanza, dalla ricerca sui genomi o sulle malattie, ma anche per tutti i settori di produzione di sapere innovativo, compresi l'industria e la pubblica amministrazione. La Grid fa incontrare

in modo trasparente i dati con le risorse di calcolo, ovunque queste siano collocate, e riporta al ricercatore/utente informazioni di tipo testo od immagine direttamente utilizzabili.

Il modello di analisi distribuito che si utilizzerà per gli esperimenti LHC, basato su Grid, è antitetico al modello centralizzato che prevede la concentrazione di tutte risorse al CERN.

1.2.2 Ambiente di sviluppo e controllo qualità

Come precedentemente visto in questo capitolo, la comunità al lavoro sull'esperimento è vasta e dispersa per il globo. Per mettere in condizione ogni singolo sviluppatore di lavorare al proprio meglio e in modo coordinato con il lavoro degli altri è stato sviluppato il *Software Development Environment*, un'infrastruttura mirata a coordinare il lavoro dei collaboratori nelle seguenti aree:

- sviluppo: un ambiente di sviluppo comune, la localizzazione uniforme e centralizzata del codice sorgente e della documentazione, insieme a regole e guide di coding, permettono di organizzare e regolare la produzione di nuovo software e il mantenimento di quello esistente;
- integrazione: offre uno schema per facilitare l'integrazione del lavoro dei vari collaboratori;
- rilascio: prevede chiare strategie per il il rilascio del codice;
- distribuzione: offre file binari e sorgenti necessari all'installazione del software in qualsiasi centro di sviluppo e/o produzione sparso per il mondo;
- controllo qualità: tiene sotto controllo la qualità del codice, previene la regressione e tiene traccia dei *bug* identificati e corretti;
- documentazione: facilita il processo di documentazione del software e garantisce che ogni unità software sia adeguatamente documentata.

Per permettere a più sviluppatori di contribuire allo stesso progetto è necessario prevedere un sistema di gestione del codice in grado di conservare vecchie versioni

dei file sorgente, che abbia un log dove memorizzare quando e perchè sono avvenute modifiche e chi è stato ad effettuarle, e che infine permetta di recuperare in modo consistente codice di autori differenti.

Concurrent Versioning e Configurazione

CVS (Concurrent Versioning System) [4] è stato scelto per la gestione del codice sorgente. Sebbene esistano alternative più moderne, CVS ha il vantaggio di essere altamente configurabile e soprattutto familiare alla maggior parte degli sviluppatori.

CVS è uno strumento open-source che consente di gestire e mantenere il codice sorgente di un progetto software. Di fatto, CVS mette a disposizione del programmatore o di un gruppo di programmatori, un sistema di archiviazione del codice sorgente relativo alle procedure che costituiscono un progetto, tenendo traccia delle modifiche apportate dagli utenti alle diverse procedure inserite nell'archivio. Uno stesso archivio CVS può gestire più progetti software contemporaneamente.

Il codice sorgente è strutturato in una serie di sottosistemi, ciascuno costituito da più pacchetti. Per poter eseguire il *committing* del codice per un particolare pacchetto, un programmatore deve essere registrato nella lista di sviluppatori per quel pacchetto. Lo stato degli sviluppatori per un pacchetto è gestito dall'amministratore del sottosistema associato.

CMS utilizza come ambiente di sviluppo SCRAM [5], che è insieme un *tool* per la gestione delle configurazioni, un sistema di distribuzione, un sistema di *building* e un gestore risorse in grado di controllare in modo trasparente sia risorse che applicazioni. Esso offre un un ambiente di sviluppo comune ed è utilizzato da anni con successo.

Per permettere ai vari sviluppatori di lavorare senza interruzioni e in modo più fluido, senza dover perdere tempo a compilare grosse porzioni di codice, viene

utilizzato un *nightly build system*, NICOS [6], un tool già adottato da altri esperimenti al CERN. Al momento l'intero software dell'esperimento CMS supporta una sola piattaforma: Linux IA32. Il porting ad altre versioni di Linux su piattaforma a 64 bit e a Mac OS X è comunque in corso.

Storage, Banda passante e Tempi di elaborazione

La misurazione dei parametri dell'evento, eseguita dall'elettronica interna al rivelatore, porta alla generazione di quelli che abbiamo visto essere i dati RAW. Ciascun evento RAW, una volta compresso, occupa circa 1MB. Poichè dei 40.000.000 di eventi al secondo ne vengono selezionati 100 da conservare a analizzare, quanto appena detto porta a concludere che i requisiti di I/O dell'esperimento sono di 100 MB/s: il rivelatore produce dati per circa 100 giorni all'anno, che corrispondono a circa 10^7 secondi, il che vuol dire che in un anno il sistema registra 10^9 eventi, ossia 1 Petabyte di dati RAW già compressi.

Riassumendo quanto detto fino ad ora le dimensioni stimate del sistema sono le seguenti (sito CERN):

- **Potenza di calcolo**: il sistema offline ha circa 10^7 MIPS di potenza di calcolo e si affida pesantemente al parallelismo per raggiungere le prestazioni desiderate;
- **Capacità**: il sistema ha un immagazzinamento robotico su nastro di diversi PB e una cache disco di centinaia di TB;
- **Throughput**: la massima banda da disco a processore è dell'ordine dei 100 GB/s, mentre quella nastro-disco e dell'ordine di 1 GB/s (32 PB all'anno), sparso su decine di dispositivi. Quando il rivelatore è attivo sono necessari solo 100 MB/s di banda nastro per immagazzinare i dati provenienti dal sistema online;
- **Processori**: circa 1400 calcolatori quad core in 160 cluster con 40 sub-farm;
- **Capacità disco**: 5440 hard disk in array di 340 dischi;

- **Capacità nastro**: 100 lettori di nastro;
- **Requisiti di potenza**: circa 400 Kilowatt;
- **Requisiti di spazio**: circa 370 m^2.

1.3 Considerazioni

Come già accennato in precedenza nel corso di questo capitolo, il lavoro che verrà descritto in questo documento fa parte del software sviluppato per il CMS Data Quality Monitoring (DQM), framework che si occupa di rendere disponibile al personale al lavoro sull'esperimento un sistema di monitoring omogeneo per quanto concerne gran parte delle attività di acquisizione dati da esso previste. Dati i lunghi tempi di attivazione prima della messa in funzione degli apparati di CMS, il lavoro, partito da un'infrastruttura già in utilizzo, è stato centrato sullo studio e lo sviluppo di soluzioni alternative a quelle utilizzate e mirato all'esplorazione delle varie opzioni disponibili, sia per quanto riguarda il lato online del framework che quello offline. Lo sviluppo è iniziato con una estensione delle funzionalità dell'interfaccia grafica, fase utile a capire il funzionamento del sistema, successivamente le attenzioni si sono riversate sull'attuazione di modifiche infrastrutturali delle comunicazioni a livello di trasporto. Una gran parte del tempo è stata poi dedicata allo studio di design alternativi, studio che ha portato all'abbandono di soluzioni adottate prima che il lavoro descritto in questo documento avesse inizio in favore di quelle ancora in uso al giorno d'oggi.

Nel corso di questo capitolo è stato esaminato come l'architettura software di CMS si inserisce nel contesto più ampio dell'intero progetto LHC, nel capitolo che segue andremo ad analizzare il dettaglio dell'architettura del software, studiata approfonditamente durante i primi mesi di lavoro.

Capitolo 2

Il software nell'esperimento CMS

2.1 Introduzione

Sin dal 1998, anno in cui è stato avviato lo sviluppo del software per l'esperimento, tutti gli algoritmi di ricostruzione utilizzati dai vari componenti del rivelatore e i tool di simulazione usati per studiare la performance di tali algoritmi sono stati implementati all'interno del framework *object-oriented* COBRA [7] che, insieme con ORCA [8], ha ricoperto completamente e con successo tutte le necessità della Collaborazione per quanto concerne simulazione e design [9] del software dell'esperimento fino al 2005.

Nel 2003 è stata completata la transizione da una simulazione di rivelatore basata su GEANT3 verso una versione object-oriented basata su GEANT4 [10], un pacchetto chiamato OSCAR [11].

Nel 2006 sono state fatte diverse modifiche all'infrastruttura del framework, ai servizi che provvede e al modello per la raccolta dati in modo da preparare l'esperimento alla fase di presa dati di LHC. L'insieme del software, adesso chiamato CMSSW [12], è costruito attorno ad un framework, un *Event Data Model* e un in-

sieme di servizi necessari ai moduli di simulazione, calibrazione e allineamento, e ricostruzione.

In questo capitolo è descritto in qualche dettaglio il framework CMSSW. Nella prima parte viene discussa l'architettura a partire da requisiti e design. Successivamente si descrivono le varie funzionalità e in particolare le soluzioni utilizzate per la visualizzazione.

2.2 Architettura software

Ad alto livello gli obiettivi del software di CMS sono i seguenti:

- fornire l'ambiente e il controllo per l'esecuzione di algoritmi di ricostruzione basati su tecniche di *pattern recognition*, fit multidimensionali etc. sui dati dei singoli eventi;
- supportare il processing "in linea" degli eventi all'interno della HLT farm (Figura. 2.1) [13], per selezionare la piccola frazione di eventi interessanti per l'analisi "off-line";
- supportare la ricostruzione dettagliata degli eventi nel Tier-0;
- supportare l'analisi e la visualizzazione dei dati così prodotti da parte dei fisici della collaborazione;
- supportare la generazione di eventi simulati necessaria per l'analisi e l'interpretazione dei dati reali.

2.2.1 Requisiti

CMS ha individuato [14] i seguenti principi fondamentali che guidano il disegno generale dell'architettura software dell'esperimento.

- **Migrazione da un ambiente all'altro**: un particolare modulo software, sviluppato e funzionante in un certo ambiente, deve essere tale da poter essere utilizzato in un secondo momento anche in un altro ambiente non inizialmente previsto;

- **Migrazione verso nuove tecnologie**: le tecnologie hardware e software si evolvono durante il ciclo di vita dell'esperimento: ogni migrazione verso nuove tecnologie deve richiedere uno sforzo localizzato in una piccola porzione del sistema, idealmente senza costringere modifiche agli algoritmi di fisica;
- **Distribuzione del software**: il software deve poter essere sviluppato da scienziati facenti parte di organizzazioni diverse e geograficamente sparse;
- **Flessibilità**: non tutti i requisiti sono noti in anticipo, quindi il software deve essere adattabile senza richiedere riscritture generali di codice;
- **Semplicità d'uso**: il software deve essere facilmente utilizzabile dai fisici, che non sono esperti in informatica e non possono dedicare troppo tempo all'apprendimento di tecniche di programmazione.

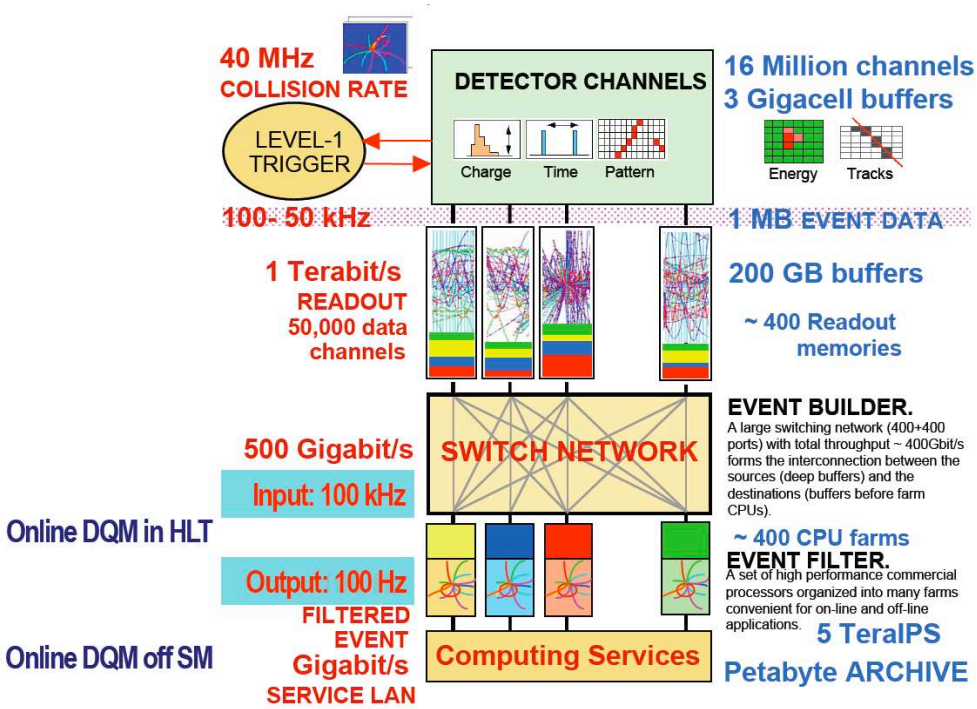

Figura 2.1: Flusso dei dati in CMS, dall'elettronica di front-end all'archiviazione su nastro.

Questi requisiti implicano che il software dovrebbe essere sviluppato tenendo presente non solo la performance ma anche la modularità, la flessibilità, il mantenimento, il controllo qualità e la documentazione. CMS ha adottato una medotologia di programmazione object-oriented e basata principalmente sul linguaggio di programmazione C++.

2.2.2 Architettura

I requisiti appena descritti portano alla seguente struttura generale del software di CMS:

- un framework con applicazioni configurabili per ciascuno degli utilizzi descritti sopra;
- algoritmi di fisica con struttura modulare, aventi interfacce ben definite in modo da essere facilmente interfacciati al framework;
- toolkit di utilità e servizio utilizzabili da ciascuno dei moduli di fisica.

Il framework definisce le astrazioni di livello più alto, il loro comportamento e i diversi modi che esse hanno di interagire fra di loro.

Esso comprende due componenti:

- un insieme di classi che rappresentano concetti specifici di CMS come ad esempio componenti del detector o caratteristiche di eventi;
- una politica di controllo che orchestra le istanze di tali classi gestendo il flusso di controllo, lo scheduling dei moduli, input/output, etc.

I moduli di fisica e utilità sono sviluppati dai gruppi responsabili ciascuno di diverse parti del detector e possono essere collegati al framework applicazione al momento dell'esecuzione. È inoltre possibile scegliere in modo semplice fra le differenti versioni dei vari moduli. La comunicazione fra moduli non avviene in modo diretto ma solo attraverso protocolli di accesso che fanno parte del framework stesso.

Per quanto riguarda i toolkit, essi soddisfano due principali categorie di servizi: i servizi per la fisica (istogrammazione, fit, algoritmi matematici, routine di calcolo di geometria e di fisica) e i servizi per il computing (accesso ai dati, comunicazione fra moduli, interfaccia utente, etc.). Essi sono basati sulle componenti dell'*LCG*

Application Area [15] e, assieme al framework applicazioni, schermano i moduli di fisica dalle tecnologie sottostanti utilizzate per i servizi di computing, in modo da facilitare l'inevitabile transizione verso nuove tecnologie che avviene col trascorrere del tempo.

2.3 Framework

L'obiettivo principale del framework CMS e dell'Event Data Model (EDM) è quello di facilitare lo sviluppo del software di analisi e ricostruzione. La facilità d'uso è una delle principali priorità in fase di design, tutti i risultati della ricostruzione devono essere resi disponibili in un formato direttamente utilizzabile per l'analisi senza bisogno di dover ricorrere ad uno strato addizionale.

Un altro modo di garantire consistenza e facilità di uso del software è l'automazione: il framework si preoccupa di registrare e gestire in modo automatico informazioni sulla provenienza di tutti i prodotti delle applicazioni evitando allo sviluppatore l'onere di doversi occupare personalmente di tale compito.

In questo paragrafo prima verrà descritto l'Event Data Model, in seguito verranno mostrate le funzionalità principali del framework per poi concludere citando i vari servizi ad esso associati.

2.3.1 Accesso all'Event Data Model

L'Event Data Model è costruito attorno al concetto di "evento", esso racchiude tutti i dati raccolti durante un *triggered physics event* assieme ad altri dati derivati. Gli eventi sono processati attraverso una sequenza di moduli che può essere specificata dall'utente. Quando un evento viene passato ad un modulo, quest'ultimo può estrarre dati dall'evento e/o inserirne di nuovi al suo interno. Quando è quest'ultima

operazione ad essere eseguita, assieme ai dati vengono anche inserite informazioni sulla loro provenienza, quindi sul modulo che li ha generati. Guardando all'implementazione, la classe Evento rappresenta tutto ciò che risulta prodotto e dedotto da un singolo *triggered readout* del rivelatore CMS. L'evento è responsabile della gestione del ciclo di vita del proprio contenuto: questo può includere oggetti che rappresentano i dati RAW del rivelatore, i prodotti della ricostruzione, i prodotti della simulazione e gli oggetti che derivano o da un evento fisico o da una simulazione del medesimo. L'evento contiene anche "metadati" che descrivono la configurazione del software usato per la ricostruzione di ciascun oggetto dati contenuto, le condizioni del rivelatore e i dati di calibrazione utilizzati per tale ricostruzione.

2.3.2 Framework - Tipi di modulo e comunicazione

Un modulo, termine generico per indicare un *worker* all'interno del framework, serve a consentire sviluppo e verifica indipendenti di diversi elementi di triggering, simulazione, ricostruzione ed analisi. Il concetto di modulo *event-processing*, ciascuno dei quali incapsula una ben definita funzionalità, relativa appunto all'event-processing, è introdotto proprio per raggiungere questo obiettivo. Tali moduli possono comunicare tra loro solo tramite l'evento, che funge da bus di dati. Ciò permette di poter testare i singoli moduli indipendentemente tra loro, e di poterli riorganizzare in sequenze diverse al momento dell'esecuzione.

La seguente è una lista non esaustiva dei tipi di modulo del framework:

- input - vengono usati per leggere i dati evento (e.g. dal sistema DAQ o da uno stoccaggio persistente) e consegnarli al framework;
- event data producer ("EDProducer") - usati nelle fasi di triggering, ricostruzione e simulazione, sono i moduli che inseriscono i dati prodotti nell'evento;
- filter ("EDFilter") - utilizzati in fase di triggering, controllano il flusso di calcolo per le liste di trigger;

- analyzer ("EDAnalyzer") - non modificano i dati evento, ma li possono utilizzare per creare istogrammi o altri eventi sommario;
- output - moduli che registrano i dati evento in una delle possibili forme persistenti.

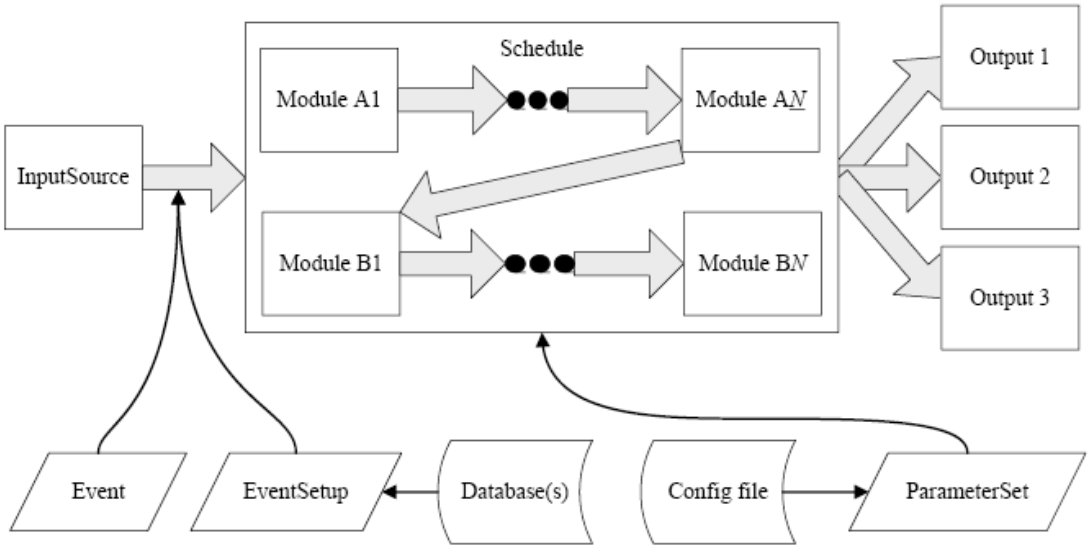

Figura 2.2: Componenti del Framework CMS ed EDM

2.3.3 Configurazione del Framework

La configurazione dell'eseguibile del framework CMS (cmsRun) è affidata ad una collezione di coppie parametro-valore, i *ParameterSet*. Essi sono creati a partire da un file di configurzione creato dall'utente. I ParameterSet usati per configurare un job saranno memorizzati nello stesso file dove risiedono i dati evento scritti da quel job.

2.3.4 Scheduler e path

I moduli sono schedulati dallo *ScheduleBuilder* e invocati dallo *ScheduleExecutor* e ciascuna loro istanza è configurata con l'uso di un ParameterSet. Essi sono i soli

ad essere configurabili. Il framework supporta due diverse tipologie di applicazione event-processing, una *unscheduled* ed una *scheduled*. Per entrambe le tipologie, gli oggetti che eseguono la ricostruzione sono istanze di EDProducer, ciascuna delle quali può essere utilizzata in qualsivoglia dei due modi detti sopra. Infatti, per entrambe le tipologie, sono utilizzate le stesse classi EDProduct e più EDProducer aventi la stessa configurazione produrranno le stesse istanze di EDProduct. Per entrambi gli stili di applicazione viene utilizzato lo stesso ParameterSet per configurare gli EDProducer e sono supportati gli stessi formati di ingresso e uscita.

2.3.5 Applicazione scheduled

Un'applicazione scheduled rappresenta la modalità base di esecuzione ed è configurata specificando un cammino (inteso come successione di moduli) attraverso il quale l'evento si muoverà.

È compito di colui che configura il job dare il corretto ordine a ciascuna successione di moduli in maniera tale da non creare conflitti. È compito invece del framework quello di ottimizzare la schedulazione di un insieme di cammini, ciascuno considerato indipendente dall'altro.

2.3.6 Applicazione unscheduled

Un'applicazione unscheduled viene configurata specificando: un insieme di EDProduct da produrre o una selezione di high-level trigger indipendenti da eseguire o un modulo di analisi da eseguire o una qualche combinazione di quanto detto sopra.

La responabilità del corretto ordinamento delle dipendenze è gestita automaticamente dal codice, che considera ogni richiesta di EDProduct come indipendente dalle altre.

2.3.7 Provenienza

Il concetto di provenienza esposto in precedenza è fondamentale in quanto è di importanza critica che gli utenti siano in grado di capire in modo univoco come ciascun prodotto della ricostruzione è stato ottenuto. Tale identificativo è costituito da un varietà di informazioni alle quali ci referiamo appunto con il termine provenienza. A ciascun EDProduct è associato un oggetto *Provenienza* che contiene questa informazione. Dove opportuno tali oggetti sono condivisi fra diverse istanze di EDProduct.

Vediamone i campi principali:

- Configurazione del modulo
  - identificatore unico che rappresenta tutti i parametri di configurazione passati a *run-time* al modulo;
  - una stringa contenente l'intero *classname* del modulo.
- Parentage

  Un vettore degli identificatori unici associati agli EDProduct, usati come input per questa parte di ricostruzione. Gli identificatori sono specifici dell'evento. Sebbene un modulo possa usare più di un input per creare il suo output, in generale non viene specificato il tipo di EDProduct al quale si riferisce ciascun elemento del vettore. Se un particolare EDProduct dovesse necessitare una tale identificazione la può memorizzare lui stesso.
- Configurazione dell'eseguibile (due componenti)
  - Una stringa, chiamata *module label*, facilmente riconoscibile, che rappresenta l'identificatore unico utilizzato per gli EDProduct creati dal modulo configurato con tale label. Questa etichetta viene creata a partire dal parametro di configurazione del modulo, ciascuno dei quali ne possiede esattamente una. Il parametro di configurazione dell'eseguibile è speciale in quanto cambiare etichetta nella configurazione causa la creazione di un nuovo modulo in quanto un unico ParameterSet determina le istanze del modulo;
  - Un numero che definisce un codice per l'intero eseguibile. Questo numero specifica le librerie disponibili durante la fase di building dell'applicazione, anche se non effettivamente utilizzate.

- Conditions Data

  Identificativo che rappresenta l'insieme di calibrazione e allineamento utilizzato per la creazione di questo EDProduct.

- Configurazione del job

  Nome fisico di processo. Un job viene messo in esecuzione in un particolare contesto (come ad esempio HLT o Reconstruction), questo nome identifica il processo sotto il quale il job è stato avviato e rappresenta solitamente una proprietà *run-time*.

Questi dati relativi alla provenienza sono distinti dai dati relativi all'evento in quanto situati in un database ausiliario, sebbene una copia possa essere resa accessibile in lettura dagli event data.

In conclusione un oggetto Provenienza racchiude le informazioni principali su come un dato EDProduct è stato creato. Ciascun EDProduct è associato (in un evento) ad una singola Provenienza.

2.3.8 Servizi del Framework

Il framework prevede due categorie di servizi in funzione del fatto che possano o meno influenzare i risultati di fisica. Ciascuna delle due categorie è gestita da un differente sistema: il *ServiceRegistry* per una e l'*EventSetup* per l'altra.

ServiceRegistry

Il sistema ServiceRegistry fornisce servizi, intesi come estensione di applicazioni, l'uso dei quali non deve avere effetti sui risultati di fisica. Esempi sono rappresentati dall'*error logger* o dal servizio di *debugging*: quali servizi devono essere utilizzati in un job e qual è la loro esatta configurazione, viene deciso attraverso il meccanismo di configurazione standard, ossia un ParameterSet.

EventSetup

Per poter processare in modo corretto un evento, ad esempio nell'analisi, sono richieste informazioni addizionali esterne all'evento stesso, come ad esempio misurazioni del campo magnetico. Questi dati si caratterizzano per avere un intervallo di validità (*Interval Of Validity, IOV*) superiore a un evento. Esistono due tipi di IOV e si distinguono dal fatto che il sistema DAQ abbia inizializzato il sistema l'intervallo di validity transition o meno: gli IOV instanziati dal DAQ (come gli eventi o un *run*) devono essere gestiti dall'*Event system* mentre di tutti gli altri se ne occupa l'*EventSetup system* che fornisce un meccanismo uniforme di accesso a tutti i dati statici (descrizione di geometrie, campo magnetico etc.) e a quelli riguardanti condizioni esterne come calibrazioni, allineamenti etc. che possono variare durante un periodo di acquisizione dati.

I principali concetti che contribuiscono alla definizione di EventSetup sono:

- Record: contiene dati e servizi aventi identici IOV;
- EventSetup: contiene tutti i record aventi un IOV che si sovrappone al *tempo* dell'evento correntemente studiato.

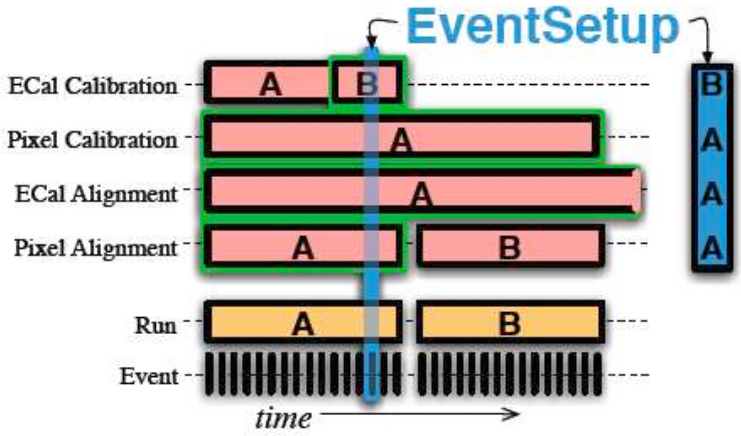

Figura 2.3: L'EventSetup è costituito dai record aventi un IOV che si sovrappone al tempo corrente studiato.

Come emerge dalla Figura 2.3, l'EventSetup garantisce l'accesso ai vari record che contiene. Se il record richiesto non dovesse essere disponibile, il framework prenderà le misure appropriate (decidendo ad esempio, di fermare l'intero job o di saltare quell'evento) in modo che, per ogni evento processato, si possa ritenere certa l'esistenza del record cercato.

L'EventSetup system utilizza due differenti componenti per svolgere i propri compiti: *ESSource* e *ESProducer*, configurate anche esse via ParameterSet.

- ESSource: una ESSource è responsabile di determinare la IOV di un record (o di un insieme di record), può distribuire dati o servizi e, sebbene non sia un requisito obbligatorio, normalmente legge le informazioni da un database;
- ESProducer: un ESProducer è, concettualmente, un algoritmo i cui input sono dipendenti dai dati in funzione degli IOV. Tale algoritmo è eseguito ogni volta che avviene un cambio di IOV per il record al quale l'ESProducer è connesso.

2.4 Event Filter

Il *Trigger and Data Acquisition System* (TriDAS) di CMS è disegnato per acquisire i dati dell'intero detector alla frequenza di collisione di 40 MHz e selezionare eventi ad una frequenza massima di $O(10^2)$ Hz per l'archiviazione e la successiva analisi. Il fattore di rigetto di $O(10^5)$ richiesto è troppo grande per essere trattato in un solo passo computazionale se si vuole mantenere alta l'efficienza, per questa ragione la selezione degli eventi avviene in due passi. Il primo passo (*Level-1 Trigger*) è disegnato per ridurre la frequenza degli eventi accettati a meno di 100 Khz, il secondo passo (*High-Level Trigger*, o HLT) è disegnato per ridurre il *Level-1 Accept rate* (L1A) di 100 Khz al valore finale di circa 100 Hz [16, 17].

Le funzionalità del sistema DAQ/HLT possono essere riassunte in quattro punti:

- esegue la lettura dell'elettronica di front-end a seguito di un segnale di Level-1 Trigger e raccoglie i dati in una singola locazione, tipicamente nella memoria di un computer;
- esegue algoritmi di selezione eventi di fisica per accettare solo quelli dai contenuti più interessanti;
- inoltra gli eventi accettati, assieme ad una piccola parte di quelli rigettati, ai servizi di monitoring online della performance del rivelatore CMS;
- provvede all'archiviazione degli eventi accettati.

Panoramica architetttura DAQ

Dalla Figura 2.4 si possono schematizzare come segue i principali elementi funzionali dell'architettura:

- *Detector Front-ends*: moduli che raccolgono dati provenienti dalle apparecchiature elettroniche;
- *Readout System*: moduli che leggono dati dai Front-end (in seguito alla ricezione di un segnale di accettazione dal Level-1 Trigger) e li immagazzinano fino a quando non saranno spediti al processore che analizzerà l'evento;
- *Builder Network*: insieme di collegamenti che provvede ad interconnettere i sistemi Readout e di filtraggio. È una *large switching fabric*, capace di sostenere un *throughput* [1] di 800 Gb/s verso il sistema di filtraggio;
- *Event Manager*: entità responsabile del controllo del data flow della Builder Network;
- Controlli: tutti i moduli responsabili di interfaccia grafica, configurazione e monitoring di sistema del DAQ;
- Sistema di Filtraggio di alto livello (High Level Trigger, HLT): insieme delle componenti per il controllo, l'inserimento dati, il monitoring, il rilevamento errori e i processori responsabili dell'esecuzione degli algoritmi HLT. Circa 500 *Builder Unit* (BU) ricevono segmenti di dati corrispondenti a un singolo evento e li ricompongono in buffer di eventi completi. Un numero appropriato di *Filter Unit* (FU) è connesso a ciascuna BU per fornire la potenza computazionale necessaria a portare avanti le funzionalità di selezione dell'High Level Trigger.

[1] Il throughput è un indice dell'effettivo utilizzo della capacità di un link, indica la quantità di dati trasmessi in una unità di tempo

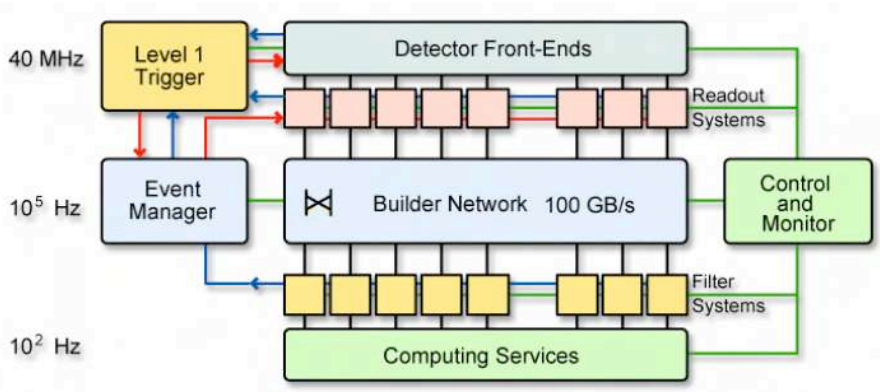

Figura 2.4: Schema di architettura generale del sistema CMS DAQ.

Requisiti HLT

Il sistema High Level Trigger deve ridurre la frequenza di eventi in uscita dal Level-1 Trigger di un fattore 10^5 per una frequenza di archiviazione totale di circa 100 Hz. Alla luminosità [2] di progetto di LHC, questo valore corrisponde ad una sezione d'urto di 10nb. Dato che la sezione d'urto di produzione del solo processo $W \to e\nu_e$ è di quest'ordine di grandezza, una parte significativa della selezione di fisica deve essere fatta online. È questo aspetto del sistema HLT che pone i requisiti più stringenti:

- il sistema deve fornire banda e risorse computazionali sufficienti a minimizzare i tempi morti a qualsivoglia luminosity, mantenendo allo stesso tempo la massima efficienza possibile per i segnali di scoperta;
- il sistema deve aggiungere agli eventi selezionati, tag contenenti le specifiche condizioni di selezione che sono state soddisfatte. Questa informazione può poi essere utilizzata offline per un ordinamento degli eventi in dataset primari;
- il sistema deve permettere la lettura, l'elaborazione e l'archiviazione di eventi necessari per la calibrazione;
- l'efficienza del sistema non deve dipendere fortemente dalle costanti di calibrazione e allineamento. Deve essere possibile in ogni caso calcolare le efficienze di trigger globali usando i soli dati, facendo riferimento il meno possibile alla simulazione;

[2] Indica il numero di collisioni per unità di area per secondo. Moltiplicata per la sezione d'urto di un processo fornisce la frequenza in eventi al secondo di quel particolare processo. Una sezione d'urto di 1 nb (nanobarn) corrisponde a 10^{-33}cm^2.

- il sistema deve essere sufficientemente flessibile da adattarsi al variare delle condizioni di run, ad esempio al degradarsi col tempo dei fasci immagazzinati nell'acceleratore;
- il sistema deve provvedere le risorse necessarie al monitoring del rivelatore CMS e fornire agli studiosi il più alto numero di informazioni (utili) possibile nel caso del verificarsi di problemi.

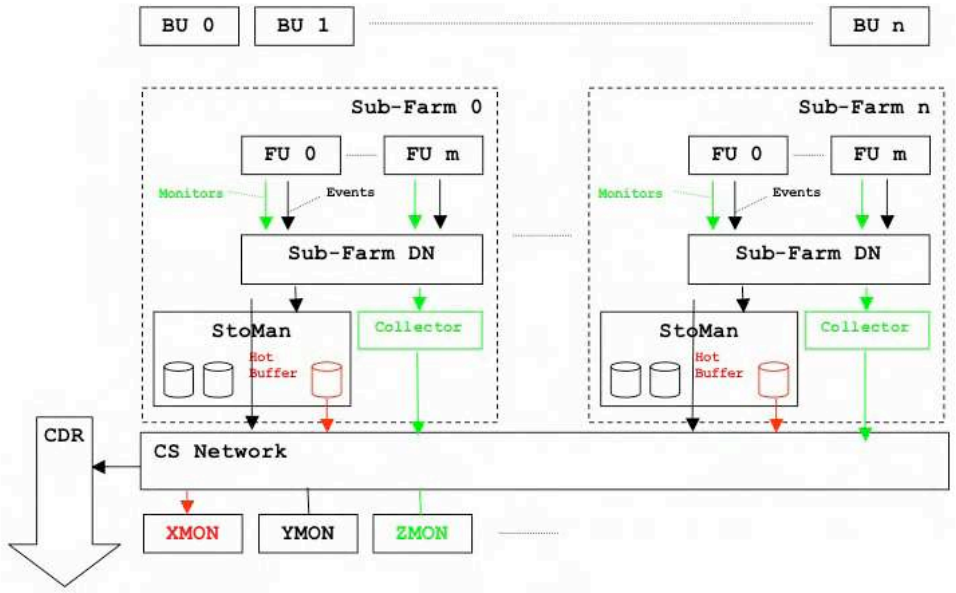

Figura 2.5: Architettura e Data Flow della Filter Farm.

Per massimizzare l'efficienza del processo di filtraggio, un evento deve essere rigettato il prima possibile: dato l'alto valore del fattore di rigetto (rispetto soprattutto a quello dei precedenti esperimenti) il sistema deve, una volta accertato il rigetto, fornire adeguate informazioni riguardo a quanto è stato scartato.

2.4.1 Architettura della Filter Farm

L'architettura della Filter Farm è descritta in dettaglio nel secondo volume del *TriDAS Technical Design Report* ed è schematizzata in figura 2.5. Basandosi sulle necessità, in termini di CPU, degli algoritmi HLT esistenti, il sistema verrà eseguito

su di una farm di circa mille calcolatori *multi-core*, a sua volta suddivisa in più *subfarm* composte da circa 100 calcolatori (Filter Unit) ciascuna.

Ogni Filter Unit si connette a un sottoinsieme di Builder Unit attraverso una rete commutata (*Filter Data Network*) capace di sostenere un traffico in ingresso proveniente dall'Event Builder avente una frequenza nominale di 100 kHz. La comunicazione BU-FU è realizzata con un semplice protocollo domanda-risposta: una FU può richiedere uno o più eventi, successivamente inviati dalle BU in modo asincrono. Non appena, all'interno di una FU, viene presa una decisione in merito ad un dato evento, quest'ultimo viene eliminato tramite un apposito messaggio inviato ad una BU. Ciascuna FU all'interno di una subfarm è assegnata a priori, tramite configurazione, ad una specifica BU, alla quale richiede servizi.

Tutte le FU in una subfarm inoltrano gli eventi accettati verso lo *Storage Manager* (SM). Il compito principale dello SM è quello di scrivere gli eventi accettati nei buffer dei dischi locali. Ciascun SM ha a disposizione spazio sufficiente per immagazzinare dati per almeno 24 ore.

2.4.2 Flusso dati

Per ciascun L1A, vengono letti dati da un totale circa 100 milioni di canali elettronici, facenti parte dei vari sottorivelatori CMS. I dati digitalizzati sono raccolti da circa 600 *Front End Driver Board* (FED), le quali operano in modo sincrono con il clock di LHC. I frammenti di evento raccolti da ciascun FED sono incapsulati in una struttura dati comune aggiungendo un header e un trailer, più marcatori all'inizio e alla fine del frammento. Le informazioni contenute nell'header includono identificatori di L1A e *bunch-crossing*, oltre alla dimensione del frammento ed informazioni di CRC (*Cyclic Redundancy Check*), utilizzate dal DAQ per verificare l'eventuale presenza di errori nel trasferimento dati. A ciascun FED è associato un

identificatore unico che permette ad ogni Filter Unit di associare ad esso un certo gruppo di canali di un subdetector in modo non ambiguo. I frammenti di evento vengono trasferiti in modo asincrono verso il DAQ attraverso un link seriale a 64-bit.

Il data flow della Filter Farm è illustrato schematicamente in Figura 2.5. Gli eventi ricostruiti sono serviti dalle Filter Unit connesse a una data BU su richiesta. Gli eventi assegnati ad una Filter Unit rimangono nella memoria della BU fino a quando non viene ricevuto un apposito messaggio di eliminazione da parte di una FU.

I compiti di una Filter Unit, dal punto di vista del DAQ, consistono nel collezionare messaggi proveniente dalla BU fino a formare un evento completo e riarrangiare i dati in memoria come frammenti FED individuali, utilizzando le informazioni presenti nell'intestazione e nel trailer. I dati sono successivamente verificati per identificare problemi di sincronizzazione o altre inconsistenze che possono essere state introdotte in fase di event building. Successivamente l'evento viene trasferito al *HLT Event Processor* che porta avanti ricostruzione e selezione. Una volta terminata l'elaborazione HLT, la DAQ libera la memoria inviando un messaggio di eliminazione alla BU.

Il *Filter Unit Event Processor*, per eseguire gli algoritmi di selezione e ricostruzione necessari a raggiungere la decisione finale HLT, utilizza moduli software basati sul framework CMS. La sequenza di operazioni che portano all'accettazione o al rifiuto di un evento è garantita essere unica e riproducibile.

L'esecuzione degli algoritmi HLT può essere monitorata facendo uso del *Data Quality Monitoring* (DQM). Tale infrastruttura verrà analizzata in dettaglio nel capitolo successivo, al momento è sufficiente aggiungere che, come desiderabile, il funzionamento di una FU può procedere indisturbato anche nel caso in cui si verifichino malfunzionamenti nel DQM.

Gli eventi accettati dall'HLT sono serializzati e arricchiti con alcune informazioni aggiuntive, prima di essere inoltrati allo Storage Manager (Figura 2.6). Tali informazioni sono sufficienti per mettere in grado lo SM di indirizzare l'evento accettato verso lo stream di uscita appropriato senza doverne esaminare il contenuto.

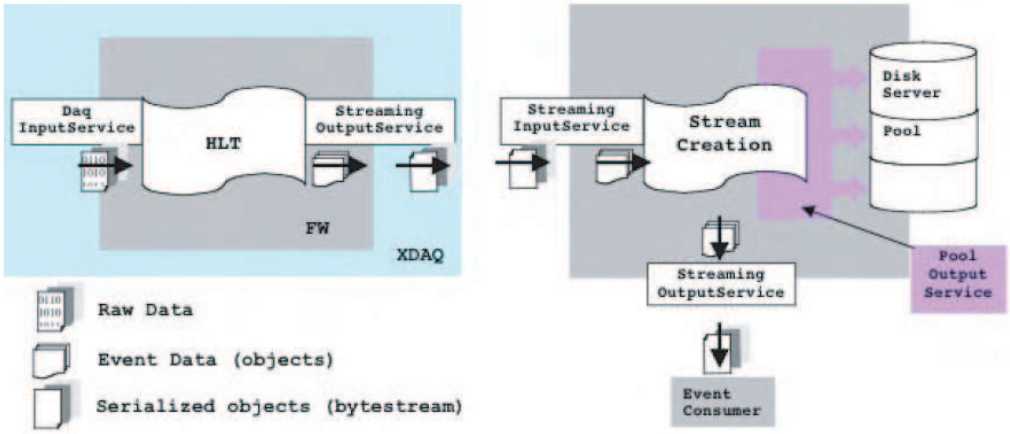

Figura 2.6: Componenti e data flow dello Storage Manager.

Lo Storage Manager è responsabile di inoltrare gli eventi accettati verso svariati predefiniti stream online, quali *disk stream* per i dati di fisica, oppure *network stream*, utlizzati dai processi che si occupano di calibrazione e monitoring. Esso è in grado di inoltrare più copie di uno stesso evento verso stream differenti: per gli stream disco vengono utilizzati formati di file supportati da *ROOT* [18]. La modalità di funzionamento di default consiste nel memorizzare direttamente in formato binario gli eventi serializzati, questo modo di operare necessita di una fase di post-processing per ricreare i file ROOT binari da trasferire alle unità di stoccaggio permanente del Tier-0. Le macchine che ospitano lo SM si occupano indirettamente anche di gestire il pool di dischi che formano il buffer online. Il pool di dischi è costituito da otto cage di dischi SATA, ciascuno con due controllers, interconnessi attraverso quattro switch *fibrechannel* con una capacità di 2Gb/s ciascuno per una banda passante su disco

totale di 2GB/s e un volume totale di 300 TB che assicurano 24 ore di operazione continua alla bandwidth di picco.

Poichè le Filter Unit processano gli eventi in parallelo, non c'è garanzia che gli eventi vengano accettati nel corretto ordine. Pertanto, nella fase di post-processing, essi vengono riordinati sfruttando le informazioni in essi contenute. La fase di post-processing ha i seguenti compiti:

- collezionare datafile relativi ad un dato stream online per tutte le subfarm;
- deserializzare gli eventi quando risultino essere in formato binario serializzato;
- riordinare gli eventi contenuti in tutti i datafile in base al loro tempo di accettazione Level-1;
- suddividere lo stream, riordinato, in sezioni di durata fissa (sezioni di run);
- creare dataset primary, ordinati in ordine di tempo in base all'*L1+HLT trigger path*;
- memorizzare questi ultimi in ROOT.

Dati RAW

Come già menzionato in precedenza, i blocchi FED di dati digitalizzati ricevuti da una FU contengono informazioni provenienti dai vari sottorivelatori, in un formato appositamente creato per minimizzare il traffico dati nell'Event Builder. I blocchi FED vengono arricchiti da un'intestazione e un trailer contenenti, fra le altre cose, un identificatore unico che permette di indirizzare il blocco al modulo di unpacking corretto: i valori di tali identificatori sono pre assegnati ai vari sotto rivelatori dal gruppo DAQ. I FED, dopo essere stati ricevuti nella FU, vengono riordinati in una struttura dati indicizzata dall'identificatore di source, preservando il loro formato interno, intestazione e trailer inclusi. Questa struttura dati (*FEDRawData*) può inoltre essere memorizzata col resto dell'evento. Ciascun modulo di unpacking, specifico per ogni sottodetector, estrae dall'evento il gruppo di FEDRawData

che corrispondono al proprio input. Successivamente procede con l'estrazione delle principali informazioni da ciascun canale del rivelatore, creando il corrispondente oggetto *digi*. La corrispondenza tra un certo elemento del rivelatore ed i suoi canali elettronici si ottiene attraverso una mappa di canali contenuta in un database. Gli oggetti digi contengono informazioni sufficienti per identificare il canale corrispondente nel contesto del framework di ricostruzione (*DetId*), così come il contenuto di basso livello (e non calibrato) di tale canale (*ADC counts*). Essi sono inoltre ordinati in insiemi, in modo da permetterne una veloce identificazione da parte di un modulo di ricostruzione.

Output service e Storage Manager

Un processo HLT in esecuzione su di una Filter Unit utilizza un modulo speciale denominato *Output Module* per serializzare gli eventi accettati da spedire successivamente allo Storage Manager. Gli eventi serializzati vengono poi passati ad un thread, sempre in esecuzione sulla FU, responsabile di mantenere il necessario handshake fra FU e il Collector in esecuzione nello SM. Il protocollo usato per la comunicazione distingue fra messaggi di tipo *control* e messaggi di tipo *data*, i primi sono utlizzati per:

- stabilire la connessione;
- controllare la consistenza delle informazioni di inizializzazione fra le due applicazioni;
- controllo del flusso.

mentre i messaggi di tipo data sono usati per spedire i dati evento da una FU allo SM. Lo SM Collector trasferisce gli eventi raccolti verso una o più code di stoccaggio, come descritto nella configurazione inserita nell'intestazione dell'evento.

2.4.3 Controllo del flusso

La Filter Farm opera come componente del DAQ, sotto la supervisione del *Global DAQ Run Control System*, quest'ultimo genera messaggi successivamente trasmessi a subfarm di PC da un *Subfarm Manager* (SFM). Il comportamento delle Filter Unit e dello Storage Manager è controllato da macchine a stati: ciascuna transizione di stato viene monitorata dal SFM e riferita al run control una volta completata. Sotto richiesta del Subfarm Manager, un processo demone in esecuzione su ciascun PC della subfarm, genera i processi XDAQ [19] necessari che gestiranno le singole componenti. Dopo che questi sono stati inizializzati e configurati sarà possibile avviare un run. Il comando di start di un run è accompagnato da un identificatore di run (*Run Number*) che potrà essere successivamente usato per identificare gli event data corrispondenti. L'HLT viene configurato solo all'avvio di un run e la sua configurazione consiste in una sequenza di moduli di ricostruzione e selezione, dei loro parametri e di tutte le costanti di calibrazione ed altri parametri necessari a portare avanti la selezione, inclusa la configurazione del trigger di primo livello (Level-1 trigger). Un run può essere terminato solo come conseguenza di un comando dal global Run Control e causa l'interruzione del trigger di primo livello e il flush dell'Event Builder. Tutta la sequenza viene portata avanti sotto il controllo del Subfarm Manager: una volta completato il flush, ciascuna subfarm della Filter Farm riceve l'istruzione di concludere il run e, prima che lo SM possa chiudere gli stream di uscita, tutte le FU devono comunicare di aver chiuso il run con successo.

Tolleranza agli errori ed eccezioni

L'insieme dei sistemi Event Builder e Filter Farm sono stati progettati per essere tolleranti agli errori, in particolare l'interruzione nel funzionamento di una Filter Unit in una subfarm comporta solo una degradazione della performance dell'Event

Builder. Poichè gli eventi non vengono eliminati dalle BU fino alla ricezione dell'apposito messaggio, che indica il successo dell'elaborazione dell'evento, non è possibile che avvenga una perdita dati a causa del fallimento di una Filter Unit.

In caso di fallimento dello Storage Manager, le FU possono passare ad una modalità di memorizzazione locale, in questo modo gli eventi accettati vengono memorizzati nel disco locale della Filter Unit così che la presa dati possa continuare indisturbata. Quanto appena detto si può verificare anche nel caso in cui i buffer disco locali risultino pieni a causa di malfunzionamenti del sistema di Data Management. Il fallimento dello SM può portare a perdite di dati nel caso in cui uno o più dei file di output risultino corrotti, od in caso di fallimento del file system che controlla il buffer locale.

I messaggi di log generati dalle componenti della Filter Farm sono raccolti dal sistema di logging del DAQ e vanno a far parte del *run conditions database*, in modo da poter essere analizzati offline per individuare e comprendere i problemi che possono venire a verificarsi negli stream di dati. I messaggi di errore generati da una Filter Unit e dallo Storage Manager vengono inoltrati al Subfarm Manager, componente in grado di analizzare e filtrare i messaggi di errore per evitare tempeste di rete. Gli errori che non possono essere gestiti localmente dal Subfarm Manager vengono reindirizzati verso il *DAQ Error Handler* il quale, a differenza del sistema di logging, garantisce il recapito di tutti i messaggi di errore.

2.5 Visualizzazione

La visualizzazione è parte essenziale dell'analisi fisica e un insostituibile tool di debug in un'ampia varietà di situazioni, come ad esempio simulazione e ricostruzione offline, analisi dei dati, *test beam* e monitoring del detector durante le fasi di commissioning

e running. Il cuore del sistema per la visualizzazione in CMS è l' Interactive Graphics Core for User Analysis (IGUANA) [20, 21] ed ovviamente è stato scelto anche per lo sviluppo dell'interfaccia utente del Data Quality Monitoring, come vedremo più avanti.

2.5.1 IGUANA

IGUANA è un progetto atto a fornire un framework di visualizzazione basato su plugin [3] ed una serie di toolkit per la grafica interattiva in 2D e 3D, con particolare riguardo verso l'*event display* e la visualizzazione interattiva della simulazione del rivelatore (Figura 2.7).

Il software IGUANA non si presenta come un programma monolitico, piuttosto come un toolkit modulare in C++, nel quale gli sviluppatori scelgono dal toolkit solo quelle parti che sono rilevanti per il loro caso specifico. Le unità principali dell'architettura di IGUANA sono:

- un sottile strato di funzioni di utilità e portabilità;
- un piccolo kernel che coordina i plugin:
  - quelle che definiremo *application personalities*;
  - una sessione principale che insieme alle estensioni formano lo *shared application state*;
  - le componenti dell'interfaccia utente: siti e browser;
  - i metodi per la rappresentazione che mappano gli oggetti dell'esperimento ai vari browser;
- software esterno importato in IGUANA, e software esterno che ne rimane invece al di fuori.

[3] Questo termine si riferisce ad un tipo di programma che si integra saldamente ad una applicazione più grande per fornire a quest'ultima speciali funzionalità.

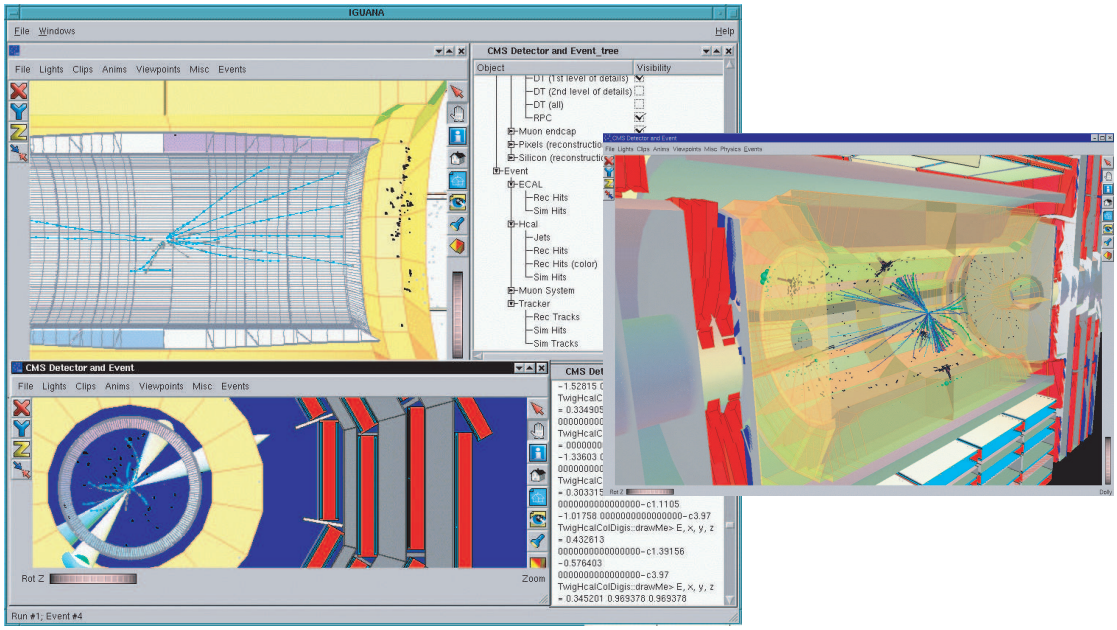

Figura 2.7: Esempi di visualizzazione del modello del rivelatore CMS e degli eventi tramite IGUANA.

Vediamo ora una descrizione del paradigma *Model-View-Controller* che è alla base dell'implementazione utilizzata da IGUANA, per poi andare ad analizzare caratteristiche specifiche di IGUANA stesso.

2.5.2 Paradigma MVC (Model-View-Controller)

Il Model-View-Controller (MVC) (Figura 2.8) è un *architectural pattern* [4] utilizzato in ingegneria del software. In applicazioni complesse uno sviluppatore tende a preferir separare i dati (Model) da quanto riguarda l'interfaccia utente (View). In questo modo le modifiche all'interfaccia non andranno a intaccare la gestione dei dati, che a loro volta potranno essere riorganizzati facilmente senza modificare l'interfaccia. Il paradigma Model-View-Controller risolve questo problema separando l'accesso ai dati dalla loro rappresentazione e interazione con l'utente, introducendo un componente intermedio: il Controller.

[4] *Software pattern* che offre soluzioni architetturali nell'ambito dell'ingegneria del software

L'applicazione deve separare i componononenti software che implementano il modello delle funzionalità, dai componenti che implementano la logica di presentazione e di controllo che utilizzano tali funzionalità.

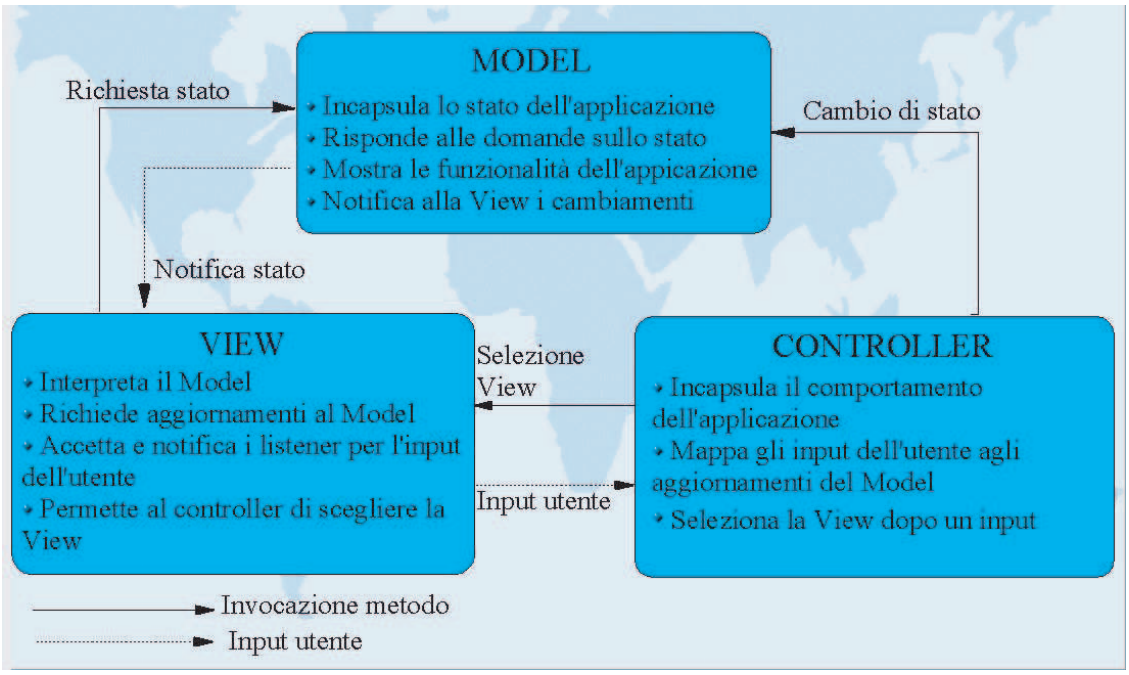

Figura 2.8: Semplice diagramma che descrive il rapporto tra Model, View e Controller. (Linee solide: associazione diretta Linee tratteggiate: associazione indiretta

Vengono quindi definite tre tipologie di componenti che soddisfano tali requisiti:

- **Model**

  Il Model è una rappresentazione dell'informazione specifica rispetto al dominio nel quale l'applicazione opera. Incapsulando lo stato dell'applicazione definisce i dati e le operazioni che possono essere eseguite su questi ultimi esponendo alla View e al Controller rispettivamente le funzionalità per l'accesso e l'aggiornamento. Il Model può inoltre avere la responsabilità di notificare ai componenti della View eventuali aggiornamenti verificatisi in seguito a richieste del Controller, al fine di permettere alle View di presentare agli occhi degli utenti dati sempre aggiornati.

- **View**

  La View visualizza il Model in una forma adatta all'interazione, tipicamente un elemento dell'interfaccia utente. La logica di presentazione dei dati viene gestita solo ed esclusivamente dalla View che deve quindi gestire la costruzione dell'interfaccia grafica mediante la quale gli utenti interagiranno col sistema. La View inoltre delega al Controller l'esecuzione dei processi richiesti dall'utente dopo averne catturato gli input, così come la scelta delle eventuali schermate da presentare.

- **Controller**

  Il Controller processa e risponde ad eventi, tipicamente interazioni da parte dell'utente, e può operare modifiche al modello. Il Controller non rappresenta un semplice ponte tra View e Model. Realizzando la mappatura tra input dell'utente e processi eseguiti dal Model e selezionando le schermate richieste dalla View, implementa la logica di controllo dell'applicazione.

2.5.3 IGUANA e il paradigma MVC

I paragrafi che seguono andranno a dare un'analisi di IGUANA e di come i concetti appena esposti hanno trovato applicazione durante lo sviluppo del framework.

Application Personalities

Quando un programma IGUANA è in esecuzione, per prima cosa crea un oggetto *sessione* all'interno del quale in seguito attacca un'*application personality*, ossia il programma principale che determina tutti i comportamenti da seguire in fase di esecuzione. Tipicamente la personalità estende immediatamente l'oggetto sessione con *servizi* pertinenti alle finalità dell'applicazione (IgExtension). È quindi la combinazione di personalità ed estensioni che forma l'applicazione: la personalità espone funzionalità installando interfacce di servizio nella sessione.

Browser e siti

Una application personality interattiva crea poi la sua interfaccia utente. Una parte importante del design è quella di creare interfacce modulari ma semplici e tali da poter essere create dinamicamente, ad esempio a partire da una descrizione testuale. Due sono le componenti di un'interfaccia: i *browser* e i *siti* (IgBrowser, IgSite). Un browser è un modo di guardare ed interagire con gli oggetti applicazione ma può anche controllare altre cose, come ad esempio un taglio nella visualizzazione di un istogramma. Esso non deve forzatamente essere grafico: può semplicemente stare in background e rispondere a richieste provenienti da altri browser. I siti ospitano browser, ad esempio fornendo una finestra per la visualizzazione di un browser 3D. In modo più generale i siti mettono a disposizione oggetti di tipo *widget* [5], da utilizzare come contenitori per altri siti o browser. Anche i siti non necessitano necessariamente di una rappresentazione grafica.

Rappresentazione oggetti

Gli oggetti possono essere rappresentati in più browser differenti. Tipicamente un browser ha un *modello* costituito da un certo numero di oggetti *reps* (IgModel, IgRep). La tendenza è di incoraggiare l'uso di rappresentazioni specifiche per modello e non gli oggetti stessi, come si vuole da una buona implementazione del paradigma Model-View-Controller. Questo permette di separare la rappresentazione dell'oggetto dall'oggetto stesso semplificando molto il design e facilitando il controllo delle dipendenze software. L'implementazione di quanto appena detto richiede che l'oggetto rappresentabile sia derivato da un tipo base comune. È possibile usare il tipo predefinito (IgRepresentable) o rendere quest'ultimo un alias per un'altra classe con il solo vincolo che il tipo sia polimorfico.

[5] Un widget è un elemento, tipicamente grafico, di un'interfaccia utente di un programma, che facilita all'utente l'interazione con il programma stesso. Ad esempio un bottone è un widget.

Comunicazioni

Le comunicazioni avvengono attraverso tre canali principali: le estensioni di una sessione, i messaggi di servizio ed i metodi di mapping degli oggetti citati in precedenza. I primi sono già stati trattati. I messaggi di servizio permettono ai browser di condividere messaggi tipo "Ho selezionato questo": tutti gli osservatori di un servizio possono mantenere insieme uno stato coerente e comunicare l'uno con l'altro senza conoscere quasi nulla di chi ha originato il messaggio. L'ultimo canale è dato dalle operazioni oggetto-rappresentazione-modello, una delle quali, ossia la creazione di una rappresentazione, è già stata menzionata. Un'altra operazione è la notifica all'oggetto sorgente di una modifica avvenuta ad una sua rappresentazione.

IGUANA è stato scelto dall'esperimento CMS come framework per la visualizzazione ed è stato usato per lo sviluppo della prima versione dell'interfaccia grafica del Data Quality Monitoring, come vedremo nel capitolo successivo.

2.6 Considerazioni

Dopo aver visto nel capitolo precedente com'è strutturata l'architettura software generale di LHC, in questo capitolo l'attenzione è stata spostata verso una descrizione dettagliata di come lo stesso problema è stato affrontato in relazione allo specifico esperimento CMS. L'analisi ha coperto tutti i campi desiderabili in una descrizione accurata: nella prima parte sono stati esposti requisiti e design generale dell'architettura, in seguito l'attenzione si è spostata sul framework, studiando tutte le varie componenti previste, le loro interazioni e il formato dei messaggi nelle comunicazioni fra componenti. L'analisi si è poi rivolta a monte, analizzando il modo in cui viene gestito e controllato il flusso dei dati, a partire dagli eventi generati in LHC, e si è conclusa con una descrizione del framework di visualizzazione, che permette ai vari

utenti sparsi per il globo di accedere e controllare efficientemente dati e componenti in modo trasparente e semplice, rispetto alle tecnologie utilizzate ai livelli inferiori.

Capitolo 3

Data Quality Monitoring

3.1 Introduzione

In questo capitolo viene descritto il sistema *Data Quality Monitoring* (DQM) di CMS. A partire da quanto discusso nel capitolo precedente vedremo per prima cosa a fronte di quali motivazioni e problematiche si è reso necessario lo sviluppo di una tale infrastruttura. In seguito, con particolare attenzione a come sono state modificate nel tempo, verranno analizzate in dettaglio le componenti del framework per poi concludere andando a coprire aspetti legati alla visualizzazione e quindi all'interfaccia grafica, di parte della quale mi sono occupato durante i primi mesi di lavoro alla tesi.

3.2 Motivazioni e problematiche

Come già detto in precedenza l'ambiente LHC impone nuovi standard dal punto di vista della mole di dati generata e quindi da trasferire. I dati relativi all'evento ed i rivelatori sono adesso molto più complessi di quanto non lo fossero nei precedenti

esperimenti di fisica delle alte energie: il numero di canali aumenta di uno o due ordini di grandezza e gli eventi di interesse sono estremamente rari (solo in un evento su 10^{12} si può osservare un decadimento del bosone di Higgs). Ad esempio, il monitoring dei 75 milioni di canali del tracciatore richiede almeno 17000 istogrammi, uno per modulo. La necessità di rendere disponibili a utenti sparsi per il globo tutte le informazioni appena descritte ha reso necessaria la nascita di un sistema di monitoring che, garantendo la qualità e l'integrità dei dati raccolti, ne permettesse un corretto trasferimento e una facile consultazione da parte degli utenti, perlopiù privi di profonde conoscenze di tecnologie informatiche. Compito del DQM, oltre a quello di garantire la qualità dei dati raccolti dalla *global data acquisition*, è quello di fornire agli utenti strumenti per la creazione, il trasporto e la manipolazione di *monitoring element* (ME), questi ultimi possono essere istogrammi a una, due o tre dimensioni, profili a una o due dimensioni, scalari (interi e numeri reali) o messaggi (stringhe). Requisito del framework deve essere quello di poter operare sia in ambiente offline che in ambiente online, lavorando quindi sia da input specificato su file sia da dati generati in tempo reale. In quest'ultimo caso è di fondamentale importanza che l'applicazione non diventi un collo di bottiglia per gli algoritmi High Level Trigger in esecuzione sulle Filter Unit, in modo da non incidere sull'efficienza della Farm e quindi del sistema in generale. Importante passo in questo senso è stato quello di disaccoppiare l'analisi degli istogrammi dalla loro produzione. Per quanto riguarda le FU è importante ricordare che, ad esperimento in esecuzione, esse (intese come sistema Farm) accettano eventi con una frequenza di 100Hz e quindi, dato che ogni evento occupa circa 1MB, hanno un volume di dati uscente di banda pari a circa 100MB/s.

3.3 FrameWork DQM

Il *Physics and Data Quality Monitoring system*, basato interamente su CMSSW e ROOT, ha come obiettivo quello di fornire un sistema di monitoring omogeneo a tutte le attività relative alla raccolta dati nell'esperimento CMS. La sua struttura deve quindi essere flessibile e facilmente personalizzabile in modo da poter essere usata senza problemi da diversi gruppi all'interno dell'esperimento. L'infrastruttura del DQM fornisce un'interfaccia generica, indipendente dalla specifica implementazione, utile alla creazione e all'aggiornamento dei ME. In generale si ha un sistema avente struttura di tipo *produttore-consumatore* [22]: i primi pubblicano una lista contenente l'informazione disponibile mentre i secondi, una volta connessi, ricevono la lista appena citata e richiedono l'informazione di loro interesse. I produttori quindi ricevono richieste di sottoscrizione a istogrammi in loro possesso e da quel momento in poi si preoccupano di fornire aggiornamenti regolari dell'informazione sottoscritta ai consumatori che ne hanno fatto richiesta. Dal lato consumatore, sono disponibili inoltre vari tool per verificare l'effettiva consistenza dell'informazione ricevuta rispetto a dati di riferimento conservati su database, aggiornare tali dati, generare allarmi, impostare filtri e creare messaggi di errore utilizzati poi dalla *central error logging facility* [1] (MessageLogger). Le problematiche descritte nel paragrafo precedente, soprattutto quelle legate al funzionamento online, hanno portato, nel corso del tempo, diversi cambiamenti alla struttura del framework. Come vedremo nel corso di questo capitolo, l'implementazione adottata inizialmente e per lungo tempo in utilizzo, a seguito di accurate analisi e test, è stata giudicata insoddisfacente, quindi, rimanendo salde le caratteristiche descritte all'inizio di questo paragrafo, la tendenza è stata quelle di migrare verso una differente architettura. Tutto ciò è avvenuto in concomitanza con il mio periodo di permanenza al CERN e quindi

[1] Servizio che raccoglie messaggi generati da vari moduli CMSSW in esecuzione nel sistema.

parte del mio lavoro di tesi è stato quello di partecipare ad un'interessante modifica strutturale che si è rivelata molto costruttiva, sia dal punto di vista dell'analisi, sia per la complessità dovuta a modularità e dimensione del software. Nella sezione seguente verranno descritte le due differenti implementazioni che si sono susseguite, prima sarà esposta la versione inizialmente utilizzata e in seguito verrà mostrato come quella correntemente approvata ed in uso al momento in cui viene redatta questa tesi differisce dalla prima descritta.

3.3.1 Implementazione via Collector

All'interno del framework, facendo riferimento alla struttura descritta in precedenza, il produttore di informazione viene detto *source*, mentre con il termine *client* si identifica il consumatore dell'informazione. Dal punto di vista dell'implementazione è importante fare notare come le due entità appena definite siano state intese come due processi distinti in esecuzione ai lati opposti del framework. Il primo (source) è generalmente in esecuzione sulle FU e, utilizzando i dati evento filtrati dagli algoritmi in esecuzione su queste ultime, si occupa di generare l'informazione di monitoring (ME) creando così il pacchetto di informazione che viene scambiato all'interno del sistema. I secondi (client) invece si trovano a valle dell'architettura e sono generalmente in esecuzione su macchine sparse per il globo: essi eseguono il lato client dell'applicazione e quindi compiti quali la visualizzazione ed eventuali operazioni di post-processing dell'informazione ricevuta. L'implementazione scelta inizialmente prevedeva anche una terza figura intermedia che facesse da tramite fra le due: il *collector*. Compito principale del collector è quello di organizzare e ridistribuire ai client l'insieme dei monitoring element da loro sottoscritto, il tutto su base periodica ed evitando ogni possibile comunicazione fra di essi e le source. Queste ultime sono definite come nodi indipendenti, esse possono avere accesso diretto ai

dati (caso offline) oppure, alternativamente, essere loro stesse a elaborare e produrre l'informazione richiesta (caso online). I client, idealmente situati al lato opposto di questa architettura, possono avere accesso ai dati tramite collector, da quest'ultimo infatti ricevono notifiche sulle informazioni di monitoring messe a disposizione dalle varie sorgenti ad esso collegate, ed è loro possibile sottoscrivere e quindi ricevere aggiornamenti periodici su un qualsiasi sottoinsieme di informazione, secondo la classica implementazione del pattern *publish-subscribe* [23].

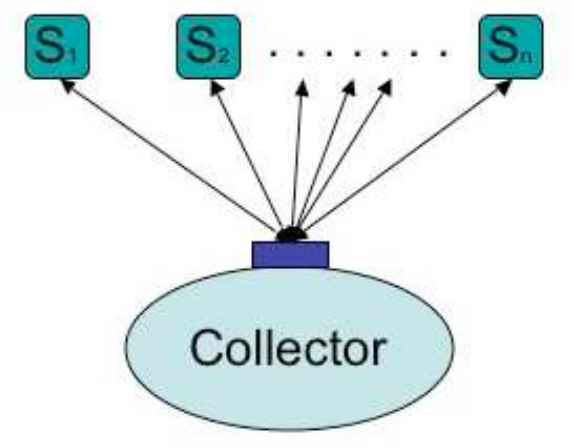

Figura 3.1: Comunicazioni collector lato sorgenti.

A fare da ponte fra questi due tipi di nodo è adibito un sistema gerarchico di nodi collector, responsabile di tutti i trasferimenti di dati fra client e source, quindi sia delle richieste di sottoscrizione, sia dell'effettivo trasferimento dei monitoring element. Poichè l'applicazione in esecuzione sulle source (software di analisi o calibrazione e algortimi HLT), soprattutto durante le fasi online, è di importanza critica, le operazioni DQM nelle source sono ridotte al minimo. Da quanto appena detto si evince come tutti i task CPU-intensive come ad esempio il confronto con ME riferimento, la visualizzazione e la copia nel database, debbano essere a carico dei processi sul lato client.

Gli obiettivi di un tale design sono quindi:

- evitare che i client si connettano alle source rischiando così di minacciare la stabilità di queste ultime o di rallentare l'applicazione principale;
- facilitare il rapido trasferimento delle informazioni di monitoring dalle source verso i collector.

La connessione tra collector e gruppi di source e client avviene tramite due connessioni distinte di tipo *one-to-many*, gestite tramite un semplice protocollo *round robin* [24]. Nel collector, lato source, i dati vengono ricevuti in modo non deterministico da parte dei nodi che hanno dati da inviare, tutto ciò avviene secondo cosiddetti cicli di monitoring [2]. Un volta terminato un ciclo il collector esegue un loop sui client connessi ed invia loro gli aggiornamenti richiesti.

In un tale design appare chiaro come la fase di produzione della *monitoring information* risulti separata dalle fasi di raccolta ed elaborazione dati. È importante notare inoltre che il DQM non dà accesso ad eventi specifici e non garantisce che due client ricevano identiche informazioni di monitoring.

Vediamo ora alcune considerazioni sulle tre componenti principali di questa architettura:

- **Source**

  I servizi del Data Quality Monitoring sono a disposizione delle source non solo per tenere traccia degli aggiornamenti ai monitoring element esistenti, ma permettono anche di applicare modifiche dinamiche alla struttura di monitoring. Un thread separato da quello principale (*MonitorDaemon*) viene periodicamente eseguito per inviare a tutti i consumatori, sempre passando attraverso il collector, la lista modificata dei ME disponibili. L'intervallo fra due aggiornamenti inviati dal MonitorDaemon, intervallo che definisce un ciclo di monitoring, può essere configurato per ciascuna source. È presente inoltre un *reset switch* per ciacun ME, utile a specificare se il contenuto di quest'ultimo debba essere resettato alla fine di ogni ciclo oppure no.

[2] Termine che indica l'invio di un'update da parte di una source, in situazioni normali di carico

- **Collector**

  Come già visto il collector serve a fare da ponte fra source e client. A differenza però di queste ultime non ha bisogno di alcuna customizzazione. Essi accettano connessioni da parte di gruppi di source e client e gestiscono tutto il traffico da questi generato tenendo traccia di tutte le connessioni attive e di tutti i dati in ingresso e in uscita. Il collector inoltre implementa anche una forma di *caching* delle informazioni, in modo da poter servire in modo efficiente connessioni dinamiche da parte di nuovi client. Più avanti, nel corso del capitolo successivo, saranno esaminati i dettagli di questo componente, dato che buona parte del mio lavoro si è concentrata sullo studio di questa architettura e sull'implementazione di modifiche atte a migliorarne affidabilità ed efficienza.

- **Client**

  In generale l'applicazione client differisce da quella source in quanto ha normalmente a che fare solo con *monitor data* e non con *event data*. Tale applicazione inoltre deve forzatamente essere customizzata prima di poter essere utilizzata da un dato sottosistema.

Da quanto detto in queste pagine emerge chiaramente come il collector rappresenti la componente più critica in questa architettura. Purtroppo, come verrà giustificato in seguito, al suo sviluppo non è stata prestata la dovuta attenzione causando quindi problemi ad un'architettura che, almeno sulla carta, appariva convincente. Il corso del tempo ha quindi portato a un progressivo abbandono della soluzione qui descritta in favore dell'integrazione con un altra implementazione già esistente, quella adottata dallo Storage Manager, che sarà adesso esaminata.

3.3.2 Implementazione via Storage Manager

Dopo varie sessioni di analisi avvenute durante i test relativi all'architettura descritta nel paragrafo precedente, in seguito a un gran numero di malfunzionamenti, primo fra tutti una generale impredicibilità nel comportamento del sistema in situazioni di stress, è stato deciso di apportare alcune modifiche allo Storage Manager e di utilizzare appunto quest'ultimo per l'adempimento di tutte le funzioni di cui si è parlato

in precedenza. Come già è stato descritto nel capitolo due, lo Storage Manager è responsabile della raccolta degli eventi selezionati dagli algoritmi HLT in esecuzione sulle FU e del loro inoltro al Tier-0, il tutto utilizzando un'implementazione basata su *server proxy* (SMProxyServer) [25]. Gli eventi raccolti sono poi resi disponibili sia per il loro salvataggio su file, sia per la visualizzazione tramite *event display* (ricostruzione online delle tracce, etc.). Per far fronte alle necessità del DQM, a tale struttura sono state apportate alcune modifiche in modo da rendere disponibili a source e client i vari dati evento. Con riferimento al paragrafo precedente appare chiaro come la figura del server proxy vada in un certo senso a ricoprire il ruolo che in precedenza aveva il collector (Figura 3.2).

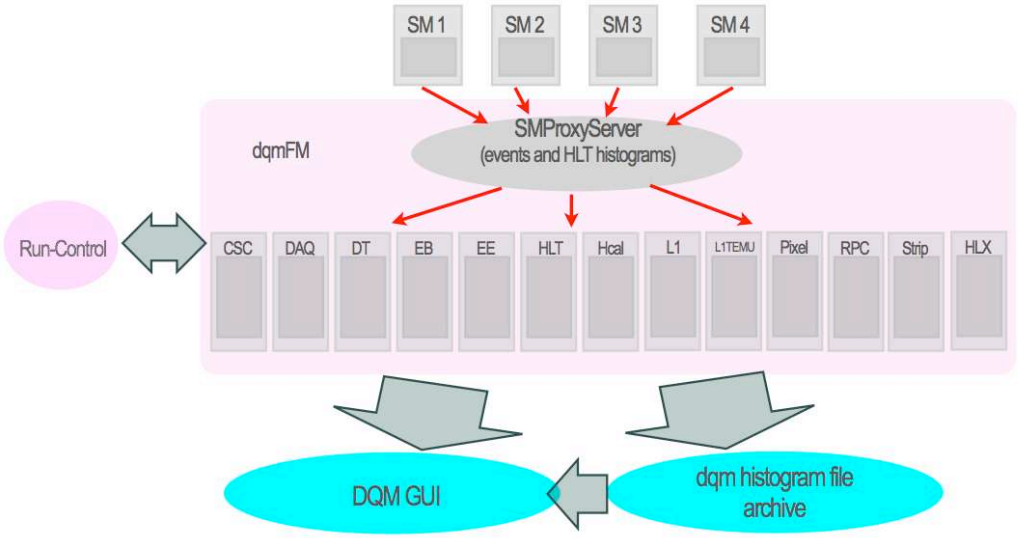

Figura 3.2: Esecuzione online del DQM tramite SMProxyServer.

È da notare inoltre che l'*event server* implementato dallo Storage Manager Proxy Server distribuisce eventi, tramite richieste HTTP, con una frequenza di 1-10 Hz [3]: a fronte di una frequenza relativamente bassa, il vantaggio maggiore di far operare il DQM attraverso il server di eventi dello Storage Manager risiede nella semplicità dell'implementazione e nella sua estrema flessibilità, oltre al fatto fon-

[3]Ricordiamo che la frequenza alla quale le FU accettano eventi è di 100 Hz

damentale che esso rende possibile sia salvare su disco dati relativi al DQM, sia poter ricostruire a posteriori le operazioni svolte, cosa impossibile con l'implementazione precedente. La modalità di funzionamento standard attualmente in uso non prevede alcun processo DQM (quindi source) in esecuzione sulle FU, queste ultime inviano eventi allo Storage Manager il quale successivamente li consegna ad un unico processo DQM che si preoccupa sia di creare gli istogrammi utilizzando gli eventi ricevuti, sia di rendere questi ultimi disponibili al framework. È inoltre possibile, nei casi in cui si rendesse necessaria una maggiore frequenza nella creazione dei monitoring element, mettere in esecuzione un singolo processo source sulle FU. Quest'ultima feature è stata resa possibile rendendo l'SMProxyServer in grado di riconoscere come input, oltre ad uno stream di eventi, anche uno stream di ME, facendolo in questo modo comportare, a livello di interfaccia, esattamente come il collector precedentemente discusso.

3.4 Visualizzazione DQM

Durante il mio periodo di permanenza al CERN non ci sono state solo modifiche architetturali al DQM, anche l'aspetto relativo all'interfaccia grafica ha subito diversi cambiamenti. L'avvicinarsi delle scadenze ha causato lo spostamento di risorse umane verso altre occupazioni ritenute più critiche, cosa che non ha permesso il continuo adattamento dell'interfaccia all'evolversi delle necessità dei sottogruppi, i quali, seppure le funzionalità dell'applicazione principale siano state mantenute nel tempo, hanno sviluppato alcune alternative, perlopiù basate sul web, che in un certo senso hanno minato l'idea iniziale di avere un approccio unificato alla visualizzazione.

Al momento della stesura di questa tesi lo sviluppo dell'interfaccia grafica del DQM è stato ripreso. La GUI sviluppata, prendendo come esempio la *custom GUI* , che sarà analizzata più avanti, sviluppata dal sottogruppo del tracciatore, è

puramente basata su web ed avente un lato server basato su QT GUI, l'event display di IGUANA e *WebTools* [26] mentre il browser sarà basato su questi ultimi.

La sezione corrente è suddivisa in quattro paragrafi principali: nel primo è descritta l'implementazione iniziale dell'interfaccia grafica per il DQM, basata su IGUANA. Successivamente, dopo aver descritto come è stata integrata all'interno del framework la possibilità di utilizzare interfacce web per la visualizzazione lato client, verrà descritta prima l'implementazione scelta dal sottogruppo del tracciatore e in seguito quella correntemente in fase di sviluppo e che già è stata brevemente citata, che si propone come soluzione globale atta a coprire le necessità generiche di tutti i sottogruppi facenti parte dell'esperimento.

3.4.1 Qt DQM GUI

L'applicazione Qt DQM GUI è stata sviluppata come plugin per IGUANA, con l'intento di soddisfare le esigenze di visualizzazione a schermo e controllo interattivo dei monitoring element sottoscritti dall'utente. Tale interfaccia, implementata utilizzando Qt e ROOT, è in esecuzione sul lato client del framework ed è composta da due thread concorrenti: uno responsabile delle comunicazioni con il DQM, l'altro dell'aggiornamento della visualizzazione. Il flusso è tale che ogni volta che il primo thread riceve un'update da parte dell'infrastruttura del DQM, aggiunge un messaggio contenente informazioni riguardo tale aggiornamento in fondo ad una coda che viene periodicamente processata dal secondo thread. Quest'ultimo preleva i messaggi dalla coda ed aggiorna la procedura di visualizzazione. Il comportamento è simile nel caso in cui sia l'utente a voler eseguire una certa azione sull'infrastruttura del DQM, esempi possono essere la sottoscrizione di nuovi oggetti o il soft-resetting [4] di

[4]Termine che indica il cleanup dei contenuti di un istogramma prima di ogni nuova update in modo da visualizzare ogni volta le informazioni relative all'aggiornamento ricevuto. Con il termine *collation* si indica invece il caso in cui ad ogni iterazione i nuovi dati ricevuti vengono sommati, ai precedenti e visualizzati in un istogramma riassuntivo.

un istogramma. Rispetto al design iniziale sono state aggiunte alcune caratteristiche significative:

- la possibilià di specificare e memorizzare una visualizzazione personalizzata degli istogrammi a schermo;
- la possibilità di decorare i monitoring element in maniera non intrusiva e dinamica.

Lo sviluppo della prima feature, dettato dalla necessità di avere diversi modi di organizzare e visualizzare i dati del DQM, è stato il primo task di mia responsabilità durante il periodo di lavoro al CERN. L'obiettivo è stato quello di aggiugnere all'interfaccia esistente la possibilità di specificare, tramite un file XML, un layout, personalizzabile a piacere (Figura. 3.3), in cui visualizzare un numero arbitrario di istogrammi secondo una disposizione su righe e colonne.

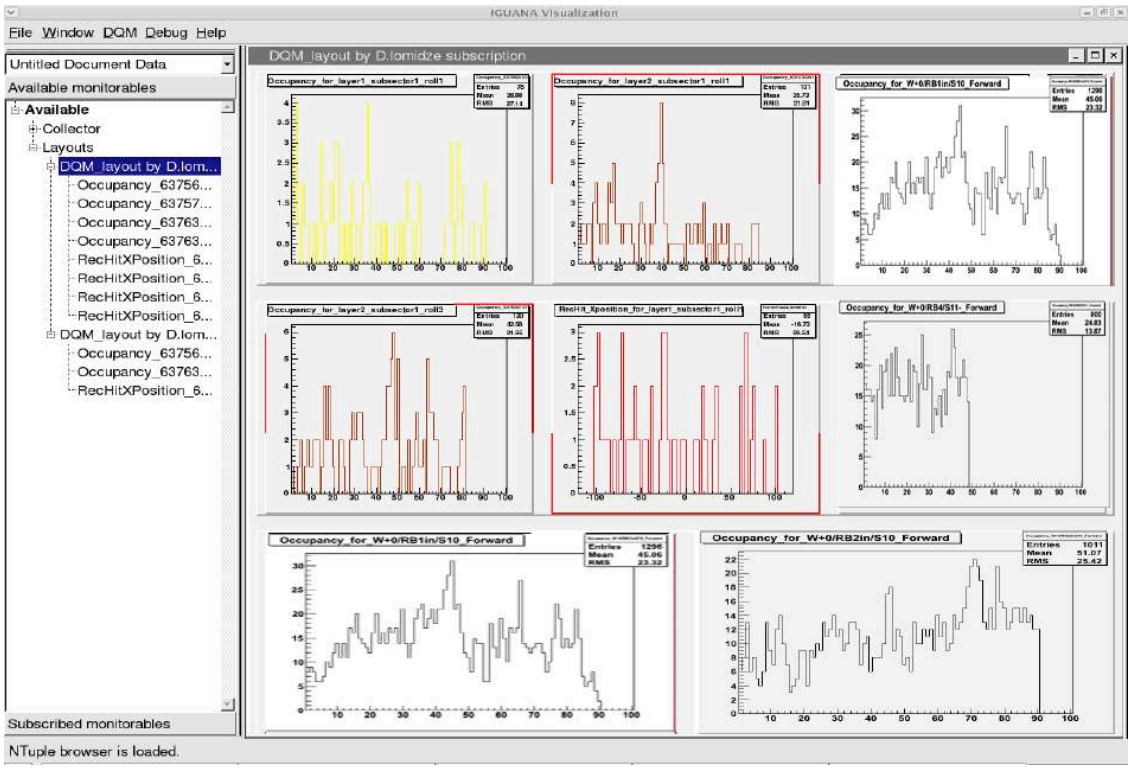

Figura 3.3: IGUANA graphical user interface: rappresentazione di un layout.

La seconda aggiunta è stata invece implementata per venire incontro alle esigenze degli sviluppatori di alcuni sottogruppi, ECAL e Tracker in particolare. Essi richiedevano un modo semplice per modificare l'apparenza di alcuni istogrammi secondo alcuni parametri specifici alla visualizzazione. Lo sviluppo di tutto questo ha richiesto la creazione di un sistema a plugin tale che, ogni volta che un monitor element soddisfa alcuni criteri predefiniti, viene attivata un'azione di *decorazione* ad essi associata che va a modificare alcune proprietà non essenziali della visualizzazione. Le due aggiunte appena descritte hanno dato agli sviluppatori dei sotto-rivelatori ampie alternative di personalizzazione. Il tutto senza dover apprendere nulla sul funzionamento di IGUANA ma semplicemente implementando una singola classe e utilizzando la loro conoscenza del framework ROOT.

3.4.2 Common web based client

In parallelo con l'interfaccia basata su Qt e IGUANA appena descritta è stata portata avanti anche una differente implementazione, questa volta basata sul web: i principali vantaggi di tale implementazione risiedono chiaramente nella semplicità d'uso e nell'accessibilità del servizio da un qualsiasi posto di lavoro nel quale sia disponibile una connessione internet. Il tutto senza dover installare componenti del framework (Figura. 3.4). Lo sviluppo si è appoggiato su XDAQ, il framework C++ sviluppato in CMS per la creazione di sistemi distribuiti per l'acquisizione dati. L'uso di XDAQ ha facilitato le fasi di disegno, programmazione e gestione delle applicazioni per la raccolta dati mettendo a disposizione dello sviluppatore un ambiente di sviluppo semplice, consistente e integrato. In prima approssimazione, un'applicazione XDAQ è in grado di ascoltare richieste HTTP e reagire invocando funzioni callback definite dall'applicazione stessa.

Quello che è stato fatto è stato quindi rendere il client DQM un'applicazione

XDAQ che, una volta eseguita nel contesto appropriato, si è arricchita di funzionalità web. In questo tipo di client DQM un thread separato si occupa di adempiere alle comunicazioni HTTP e chiamare le funzioni appropriate a seconda del messaggio HTTP ricevuto. Questo è ottenuto tramite l'implementazione di una piccola libreria *javascript* [27]. Quest'ultima mette a disposizione dell'utente una pagina web dalla quale è possibile controllare lo stato del client ed eseguire operazioni base quali avviare e interrompere i cicli di update. Il sistema è stato programmato in modo tale da rendere estremamente semplice la personalizzazione dell'interfaccia web. È infatti possibile definire funzioni *callback* sul lato client e farle corrispondere a ben determinate richieste HTTP. Tutto questo aggiungendo poche righe alla libreria predefinita.

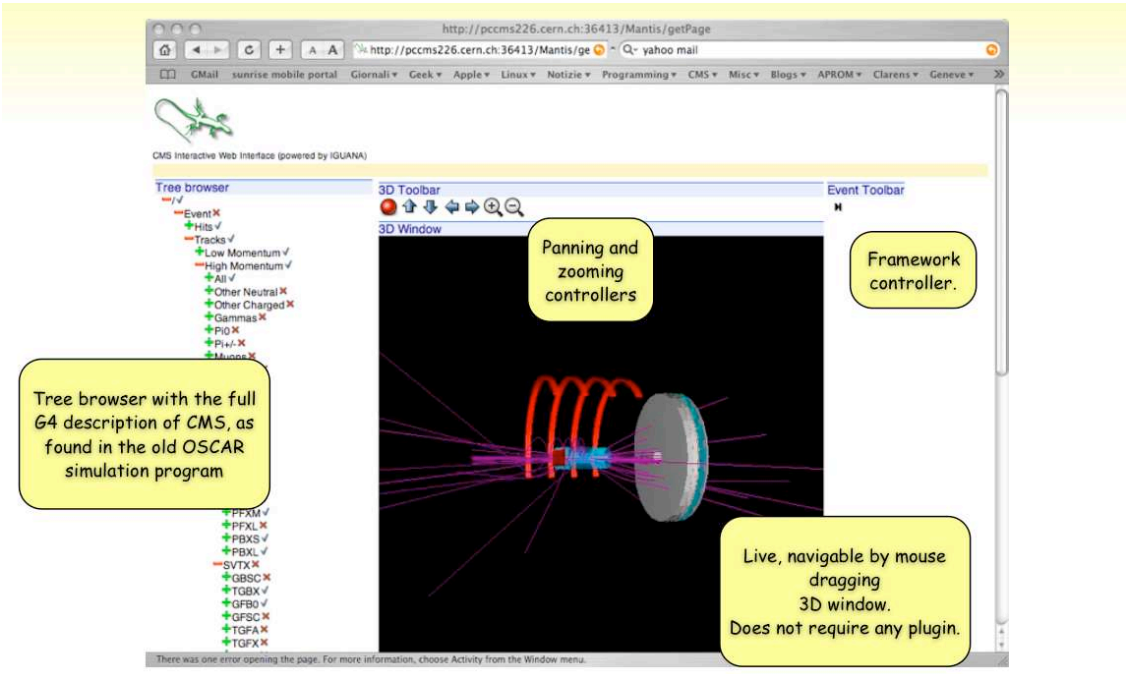

Figura 3.4: Common web based client, visualizzazione geometrie.

La possibilità di utilizzare oltre a javascript, le applet java, ad integrare il codice Html, permette inoltre di raggiungere alti livelli di complessità nel disegno e nelle funzionalità dell'interfaccia. Infine, per ridurre al minimo i tempi di attesa, in

modo da rendere l'applicazione online di velocità paragonabile ad un'applicazione standard, si è fatto largo uso della tecnologia *AJAX* [28], discussa nella sezione seguente.

AJAX

AJAX è un acronimo per Asynchronous Javascript and XML e rappresenta, più che un linguaggio di programmazione, una tecnica di sviluppo web per creare applicazioni web interattive. Nel web tradizionale (Figura. 3.5), le comunicazioni fra client e server sono completamente sincrone, quindi, maggiore è la complessità della pagina visitata, maggiore sarà il tempo da attendere prima di poter interagire con essa.

Il concetto alla base di AJAX è invece quello di gestire lo scambio in background di piccoli pacchetti di dati con il server in modo da non dover ricaricare l'intera pagina una volta che viene fatta una modifica.

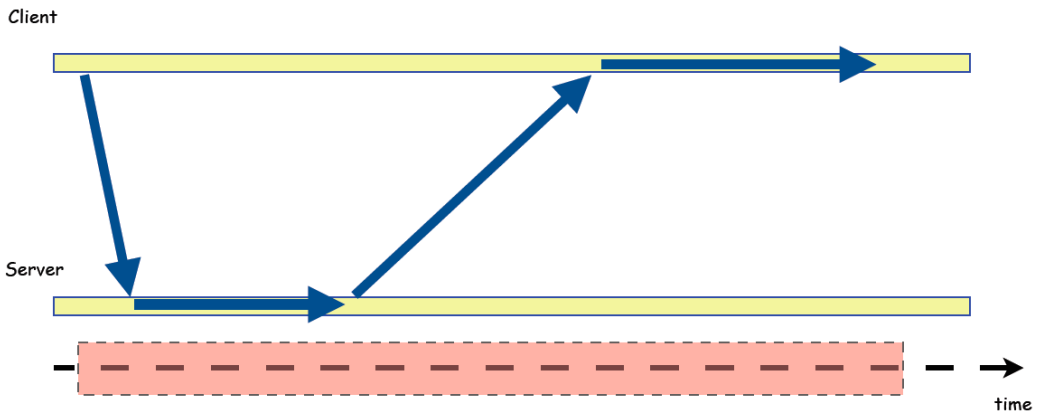

Figura 3.5: Web tradizionale: se la pagina web è sufficientemente complessa siamo costretti ad attendere il completamento di una transazione prima di poter avviare la successiva.

Infatti in questo caso il server (Figura. 3.6), alla ricezione della prima richiesta, inoltra al cliente una pagina contenente il minimo volume di dati necessario alla

visualizzazione. A questo punto se (e solo se) l'utente richiede ulteriori informazioni, queste vengono richieste sotto forma di una richiesta di tipo *XMLHTTPRequest*.

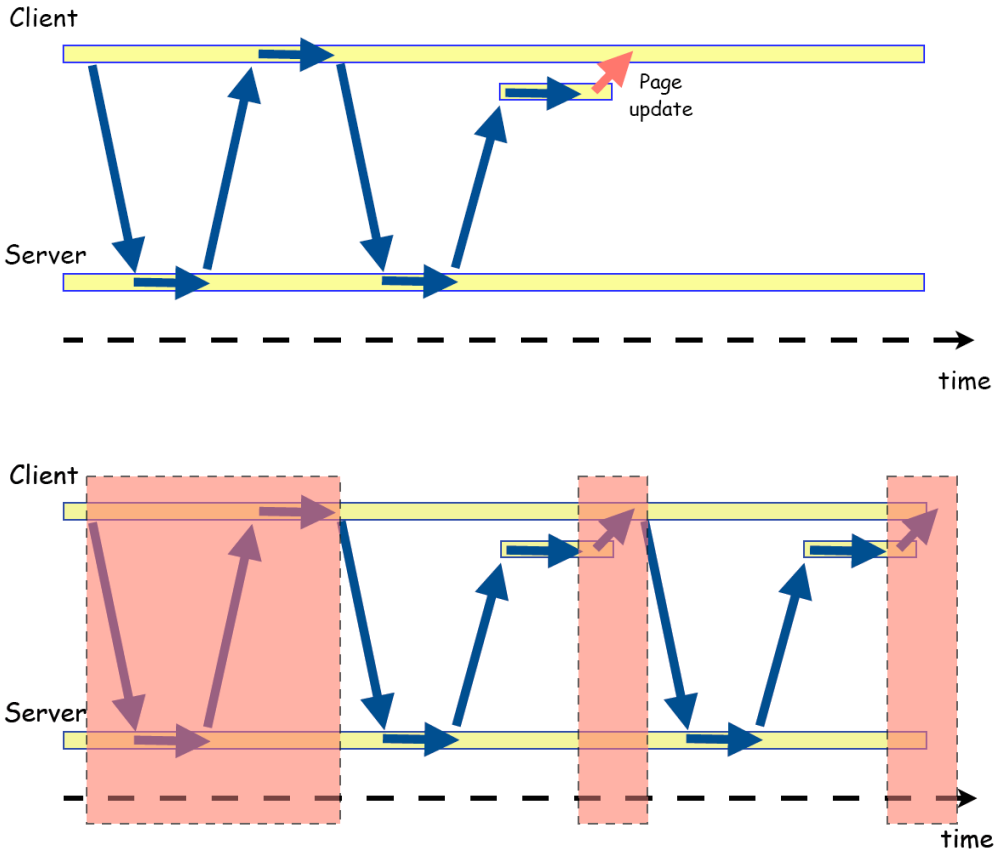

Figura 3.6: AJAX: ad ogni richiesta di nuove informazioni vengono inviate dal server solo le update strettamente necessarie, l'elaborazione di queste ultime tramite javascript garantisce l'asincronia.

Il server allora elabora la richiesta ed e produce una risposta XML contenente solo le informazioni che sono cambiate rispetto alla risposta precedente, questa risposta viene elaborata in maniera asincrona utilizzando javascript e finalmente la pagina viene aggiornata. È immediatamente chiaro il vantaggio di AJAX, ossia il fatto che l'intervallo di tempo durante il quale la pagina non è disponibile viene ridotto drasticamente (Figura. 3.7).

Le tecnologie utilizzate sono le seguenti:

- HTML (o XHTML) e CSS per il markup e lo stile;
- DOM (Document Object Model) manipolato attraverso un linguaggio come javascript per mostrare le informazioni ed interagirvi;
- XMLHTTPRequest per l'interscambio asincrono dei dati tra il browser dell'utente e il webserver;
- XML, generalmente usato come formato di scambio dei dati.

Concludendo, i pregi maggiori di questo tipo di implementazione risiedono, per quanto riguarda il lato utente, nella semplicità d'uso e di installazione, mentre l'unico vero difetto risiede nel fatto che non è possibile interagire direttamente con i *plot* mostrati a schermo.

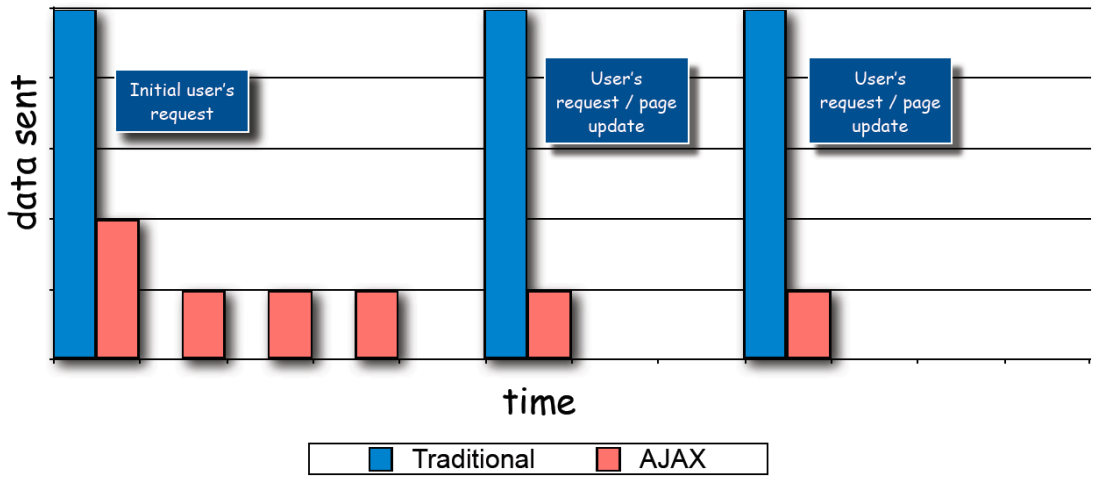

Figura 3.7: AJAX: confronto indicativo dei tempi di risposta rispetto ad un'applicazione web tradizionale.

Dal punto di vista dello sviluppatore, tenendo conto anche della mancanza concreta di personale avente buone conoscenze informatiche, l'utilizzo delle tecnologie descritte in precedenza permette di ottenere buoni risultati anche affidando lo sviluppo a personale non esperto come invece richiedono entrambe le altre due soluzioni qui esaminate.

3.4.3 Interfaccia DQM del Tracciatore

Vediamo adesso, come esempio, l'implementazione del "custom GUI" per il tracciatore di CMS. Il sottogruppo del tracciatore, sebbene soddisfatto dall'utilizzo dell'I-GUANA Qt GUI, ha dedicato parte delle proprie risorse interne all'implementazione di un'interfaccia grafica, basata sull'approccio web già descritto, che si adattasse alle esigenze interne e specifiche alla parte del rivelatore in esame (Figura 3.8).

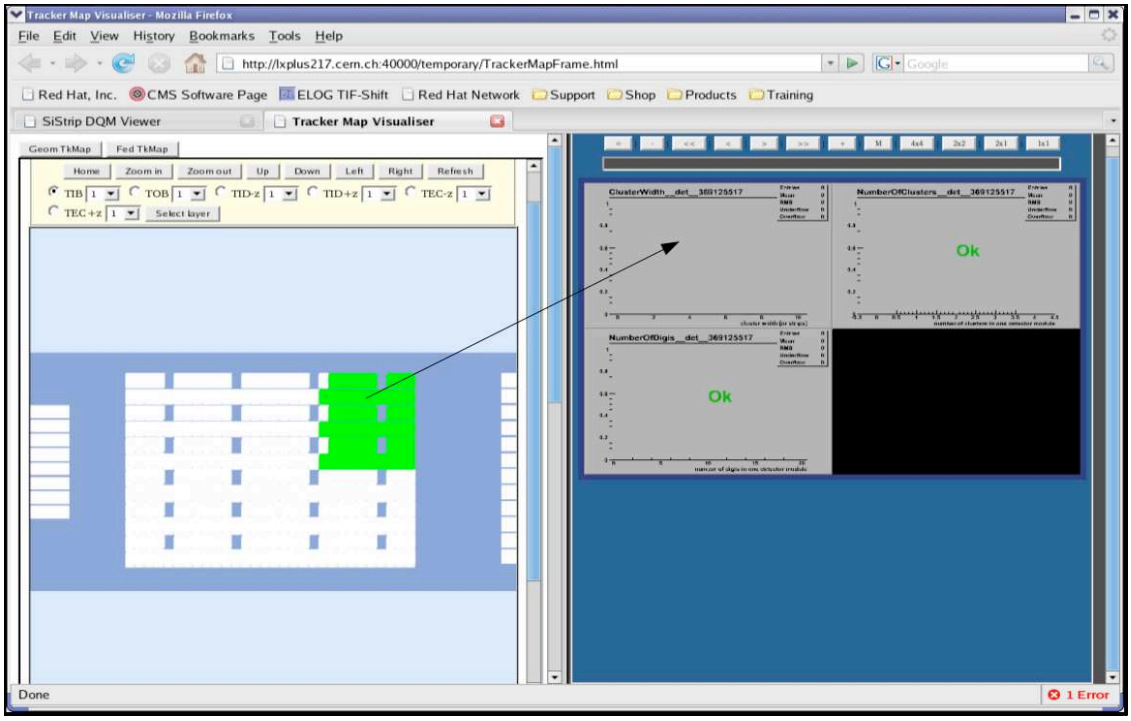

Figura 3.8: Immagine dell'interfaccia web per il DQM realizzata dal sottogruppo del tracciatore.

L'interfaccia web è stata arricchita e personalizzata per soddisfare le esigenze proprie del sottogruppo ed inoltre si è fatto un uso importante di AJAX che, come descritto in precedenza, si pone come substrato compreso fra l'interfaccia web e il web server, occupandosi di gestire in modo intelligente il flusso di richieste HTTP inoltrate dall'utente durante l'interazione con la pagina, riducendo al minimo i tempi di attesa, rendendo così l'esperienza quasi paragonabile a quella che si ha interagendo con un'applicazione standard.

La decisione di implementare una propria interfaccia è stata presa per due motivi principali:

- l'IGUANA Qt GUI, essendo pensata per soddisfare un generico utente, rappresenta in un certo senso un prodotto a "scatola chiusa" che, in quanto tale, non è personalizzabile a seconda delle esigenze ed è privo di caratteristiche specifiche e utili agli specialisti;
- Un esempio di cliente con interfaccia web basato su XDAQ ed istruzioni per creare interfacce personalizzate, esistevano già come parte dell'infrastruttura standard.

Basandosi sull'esempio appena citato, il sottogruppo del tracciatore ha creato la propria interfaccia utente integrando in essa l'utilizzo di AJAX, con tutte i vantaggi che ne derivano.

Ad oggi, con l'esperimento attivato, tale interfaccia viene utilizzata con successo e presenta il solo problema di essere disponibile esclusivamente durante la presa dati. Il webserver infatti è rappresentato dal processo client stesso, e tale processo è attivo solo quando esiste un flusso di dati online.

3.4.4 IGUANA DQM web

L'interfaccia grafica implementata per il tracciatore appena discussa mostra quanto l'implementazione basata su web possa rivelarsi attraente. Il problema principale di una tale implementazione risiede però nel fatto che solo pochi sottogruppi hanno a disposizione personale che possa impiegare tempo e risorse nello sviluppo di una soluzione analoga. Bisogna inoltre dire che le esigenze di personalizzazione non riguardano tutti i sottogruppi, molti di essi infatti si sono comunque dichiarati soddisfatti dall'utilizzo della IGUANA Qt GUI che, sebbene mostri problemi nell'utilizzo *online*, lavora molto bene in locale.

Un'ottima risposta al problema della visualizzazione è stata l'interfaccia *IGUA-*

NA DQM web (Figura 3.9), sviluppata a partire dalla primavera 2007 ed utilizzata con successo, correntemente, nell'esperimento. Essa è basata su un server web standard ed è il risultato combinato dell'esperienza di sviluppo del common web client visto in precedenza e del lavoro di customizzazione svolto per il tracciatore.

Figura 3.9: Immagine dell'interfaccia web IGUANA DQM.

Il rilancio di una soluzione di questo tipo è stato voluto con l'obiettivo di offrire un'interfaccia DQM unificata per la visualizzazione dei dati, sia online che offline, che fosse consultabile al P5 [5] , dal CERN e da remoto. In questo ultimo caso, il web server standard viene utilizzato per gestire l'interazione con l'infrastruttura e

[5]P5, point 5 è il punto, all'interno di LHC dove è collocato il rivelatore di CMS

in generale per svolgere tutte le operazioni CPU-intensive. Nel server, un certo numero di backend provvede l'accesso ai dati prodotti dal DQM, sia online che offline (Figura. 3.10). L'utente necessita quindi solo di un computer con accesso alla rete e di un web browser, e non deve installare alcun software specifico. Lo sviluppo degli elementi di backend ha permesso di riutilizzare le parti Qt-indipendenti del precedente codice IGUANA. Il server web usa, come già detto, la tecnologia Web-Tools e viene operato a partire da un nodo principale, a cavallo fra i domini *.cms* e *cern.ch*. Il sistema riceve istogrammi dai vari elementi di backend e gestisce tutte le problematiche relative alla loro raccolta e visualizzazione.

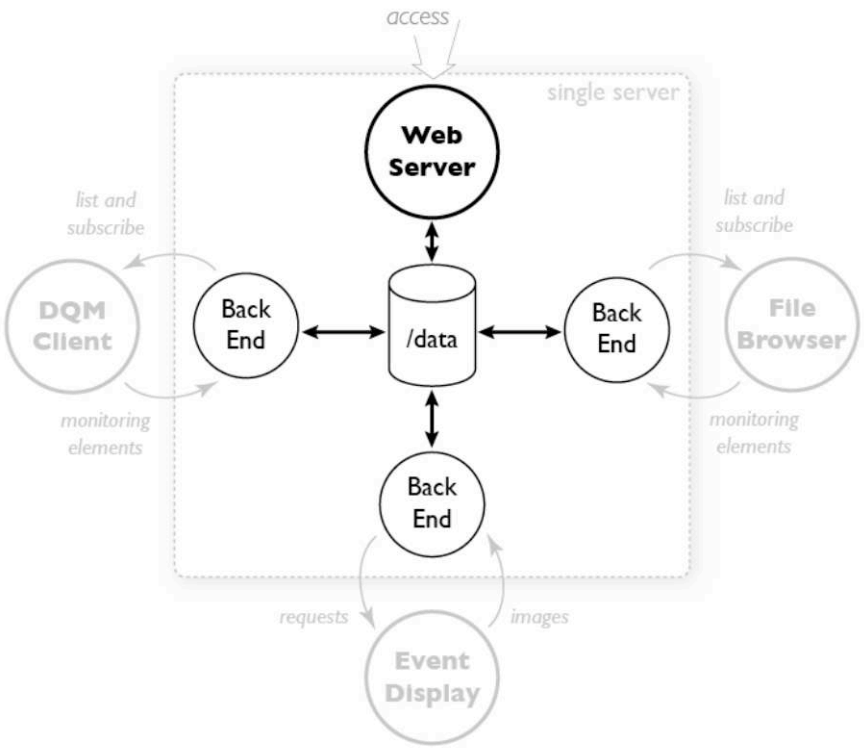

Figura 3.10: Visualizzazione DQM basata su webserver.

Guardando all'implementazione, le varie backend altro non sono che dei semplici processi *single thread* che servono a mettere in comunicazione la varie parti del framework con il server web, permettendo accesso sia ai dati del DQM che a quelli dell'Event Display e dello stoccaggio fisso. Agendo in questo modo si risol-

ve il principale problema visto con la soluzione precedente: poichè il server web è un'applicazione separata (e non più legata al processo client come visto prima), esso può essere utilizzato in qualsiasi situazione, così facendo l'utilizzo dell'interfaccia, e quindi l'interazione con i dati, è permesso sia in modalità online che in modalità offline (ad esempio nei casi in cui si volessero analizzare run già terminati), senza dover ricorrere ad altre applicazioni.

3.5 Considerazioni

In questo capitolo si è data una panoramica completa delle parti del framework DQM, sulle quali si è concentrato tutto il mio lavoro di implementazione e gran parte dell'impegno riguardante lo studio di architetture alternative a quelle in utilizzo al momento del mio arrivo nel gruppo. Si può concludere che, da un punto di vista macroscopico, i cambiamenti fondamentali siano stati due: il passaggio dall'architettura basata su Collector a quella basata sullo Storage Manager e l'evolversi dell'interfaccia grafica. In entrambi, sia da un punto di vista implementativo che puramente teorico, il lavoro svolto è stato parte attiva degli effettivi cambiamenti avvenuti. Avendo chiari questi punti è quindi adesso possibile suddividere il mio lavoro in tre aree principali, ciascuna ben differenziabile dalle altre ma nel complesso tutte altrettanto utili nello stesso contesto di sviluppo e supporto al Data Quality Monitoring.

- La prima relativa allo sviluppo vero e proprio dell'IGUANA Qt GUI per il DQM. Il lavoro è confluito nello sviluppo di un plugin per tale interfaccia, utilizzato per lungo tempo dai sottogruppi dell'esperimento prima dello sviluppo del Common web based client. Tale plugin viene discusso nel successivo capitolo. Sebbene in tempi recenti l'utilizzo dell'IGUANA Qt GUI sia stato deprecato e sostituito dalle versioni web, il mio codice è stato comunque riassorbito nelle nuove implementazioni.

- La seconda area riguarda lo studio approfondito delle comunicazioni fra moduli all'interno del DQM ed ha visto la completa riscrittura di tutte le comunicazioni fra componenti a livello di trasporto (TCP/IP). Durante lo stesso periodo di sviluppo sono stati diversi i test svolti sull'architettura per identificarne i punti deboli. Tali test, tutti implementati dal sottoscritto, sono stati inseriti nel sistema in modo trasparente, e sono serviti per dimostrare la necessità di implementare modifiche in favore di un sistema più semplice, efficiente e quindi affidabile.

- La terza area infine comprende tutto ciò che non è stato implementato, ma che è stato proposto e discusso con il gruppo per risolvere i problemi relativi sia alla prima che alla seconda area. Saranno quindi presentati in seguito alcuni modelli da me proposti come alternativa alle implementazioni usate.

Nei due capitoli che seguono verranno esaminate queste tre aree e descritto il lavoro che è stato fatto. Le prime due aree sono l'oggetto di trattazione del capitolo 4 mentre i modelli teorici formano l'oggetto del quinto e conclusivo capitolo della tesi.

Capitolo 4

Modifiche all'infrastruttura del DQM

4.1 Introduzione

In questo capitolo vengono descritte in dettaglio tutte le modifiche all'architettura software del DQM da me apportate durante il mio periodo di lavoro presso il gruppo del Data Quality Monitoring. Per facilitare la comprensione globale delle problematiche da affrontare, e quindi l'inserimento all'interno del gruppo, lo sviluppo è cominciato andando a modificare ed arricchire l'interfaccia grafica IGUANA Qt utilizzata dal DQM con nuove funzionalità. Questa prima fase ha permesso di arrivare a una comprensione dell'intero sistema, necessaria per affrontare le problematiche relative alla GUI, che investono tutte le parti del sistema stesso.

Una volta completato questo primo step si è presentata l'opportunità di scegliere come procedere con il lavoro: una prima alternativa offerta è stata quella di continuare a lavorare sull'interfaccia grafica, curandone sia gli aspetti online che quelli offline, andando così a coprire un vuoto che abbiamo visto nel capitolo prece-

dente essere stato colmato solo diversi mesi dopo. La seconda alternativa proposta è stata quella di addentrarsi più in dettaglio nel funzionamento del sistema, rendendo il lavoro in un certo senso meno implementativo ma in generale molto più vario e ricco di contenuti eterogenei; proprio per questi motivi, la scelta è ricaduta su questo secondo punto.

4.2 Layout plugin per l'IGUANA DQM Qt GUI

Per rendere più chiara la trattazione facciamo adesso alcune semplificazioni rispetto al formato dei dati DQM. Rivediamo anche le interazioni fra source, collector e client (Figura 4.1) già argomento di trattazione nel capitolo precedente.

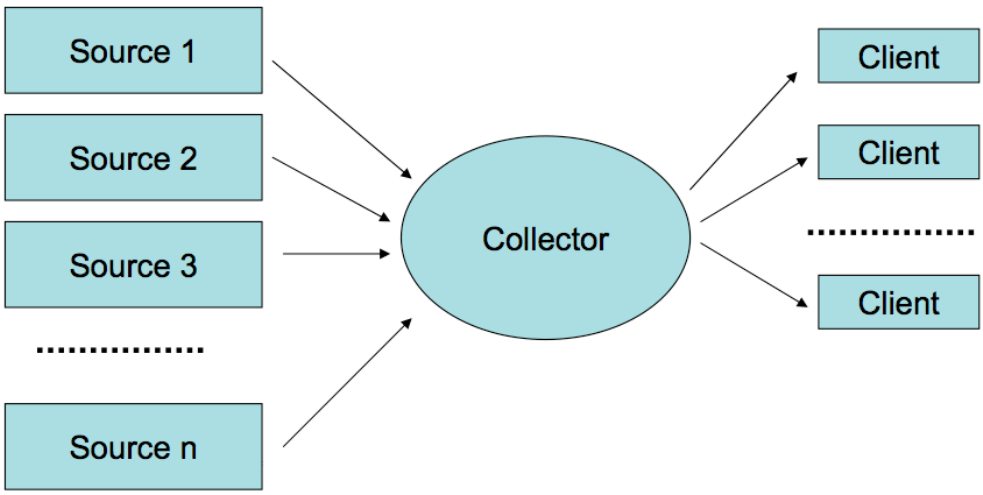

Figura 4.1: Schema Source Collector Client.

I dati DQM sono essenzialmente istogrammi serializzati in un buffer, a sua volta inserito in un messaggio che rappresenta l'oggetto scambiato fra le componenti del sistema.

Vediamo le principali interazioni fra source, collector e client:

- il collector attende nondeterministicamente connessioni da parte di source o client;
- una source, una volta connessa al collector, lo informa sugli istogrammi disponibili;
- un client, una volta connesso al collector, viene aggiornato da quest'ultimo con la lista degli istogrammi complessivamente disponibili. Questo elenco viene mostrato all'utente il quale, in un secondo momento, potrà decidere di sottoscrivere una richiesta relativa ad una certa informazione;
- da questo momento in poi le source in possesso di un'informazione sottoscritta da qualche client potranno iniziare i cicli di monitoring inviando quest'ultima al collector, il quale si preoccuperà di inoltrarla ai client che ne hanno fatto richiesta.

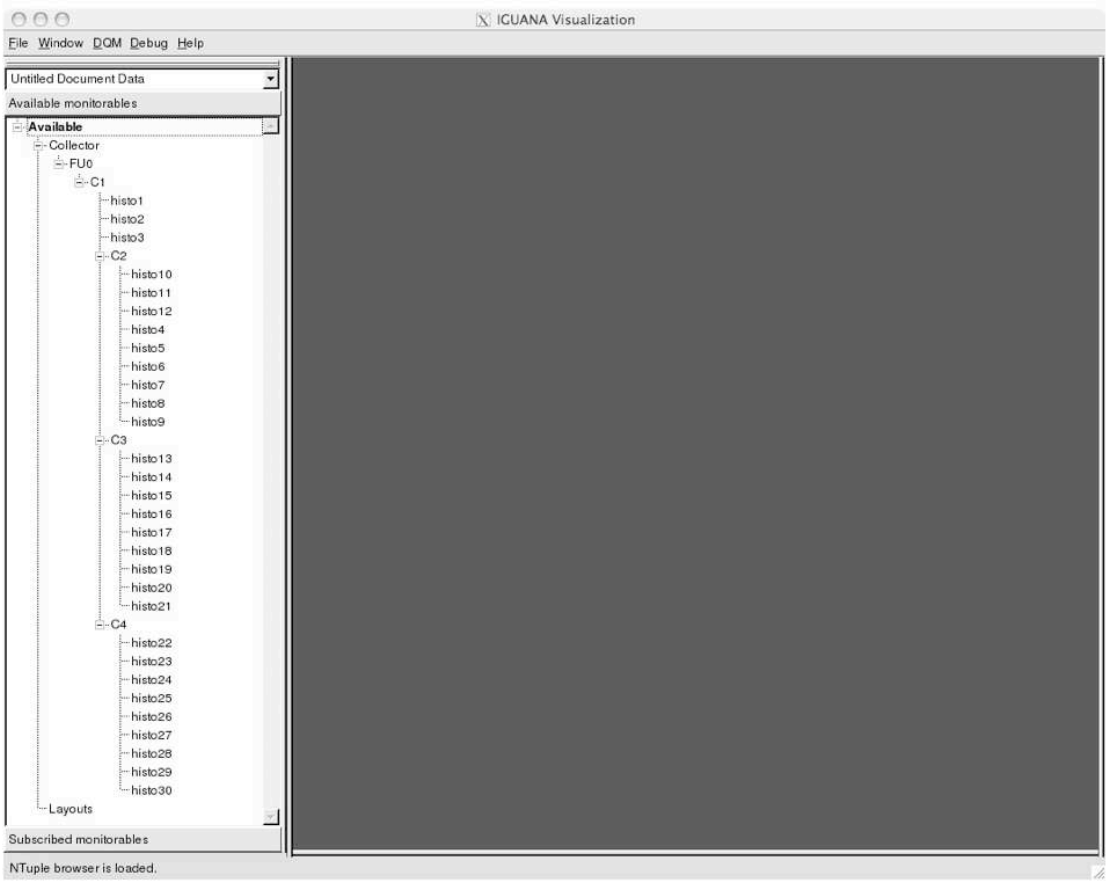

Figura 4.2: Nel box a sinistra la struttura ad albero contenente le informazioni sottoscrivibile. Il box grande a destra è vuoto in quanto nessuna richiesta è stata fatta.

Essendo l'interfaccia grafica un processo in esecuzione sul client avremo due stati principali per quanto riguarda il suo funzionamento: il primo stato, successivo alla ricezione della lista contenente l'informazione disponibile, consiste nella rappresentazione a schermo di quest'ultima tramite una visualizzazione ad albero (Figura 4.2). La navigazione di tale albero permette poi di individuare facilmente l'informazione desiderata e farne richiesta di sottoscrizione, una volta inviata quest'ultima, gli istogrammi ricevuti verranno visualizzati nell'apposita finestra (Figura 4.3).

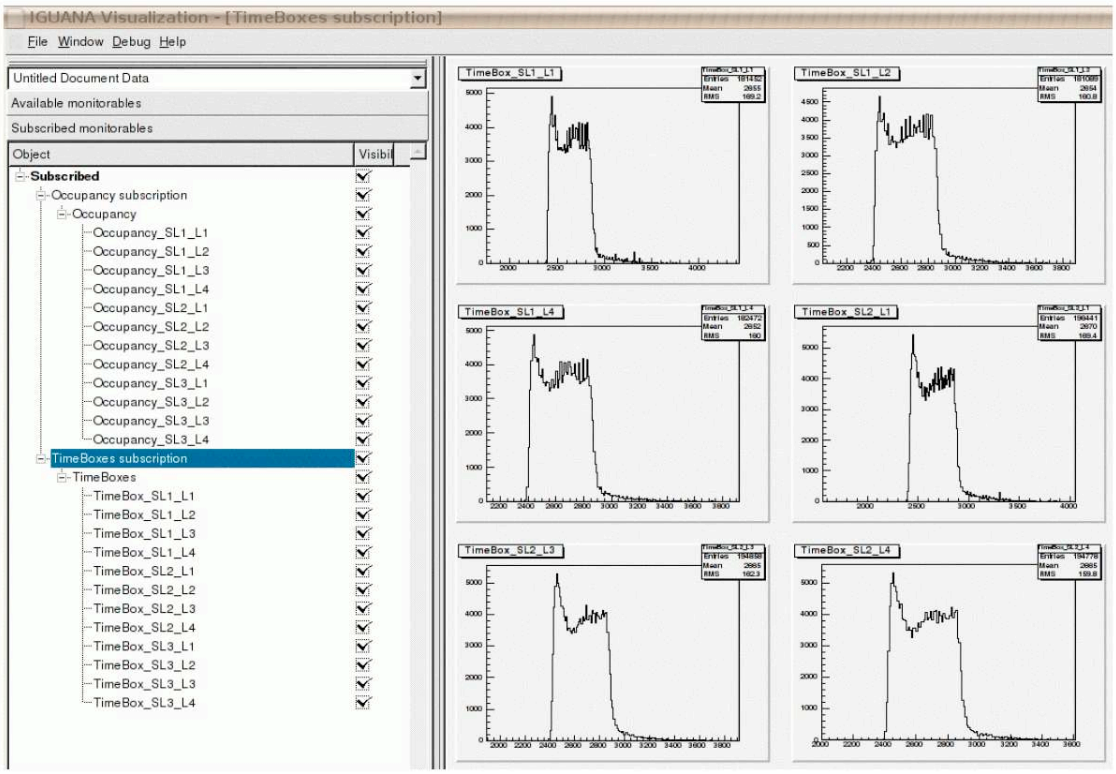

Figura 4.3: Nel box a sinistra la struttura ad albero contenente l'informazione disponibile. Nel box a destra vengono visualizzati gli istogrammi per i quali il client ha fatto richiesta di sottoscrizione.

Venendo ora alla parte di implementazione, una limitazione di tale interfaccia grafica consiste nel fatto che l'informazione sottoscrivibile, e quindi visualizzabile, è limitata o al singolo istogramma oppure a tutti gli istogrammi contenuti in una directory. Non era quindi possibile sottoscrivere istogrammi contenuti in cartelle diverse fra loro senza dover sottoscrivere l'intero contenuto di tali cartelle, poichè il numero

di istogrammi in una cartella può essere grande e dato che l'informazione cercata puó essere annidata in profondità, è chiaro come l'intero processo possa risultare poco efficiente. Oltre a questo non era possibile collocare gli istogrammi sottoscritti sullo schermo a proprio piacimento, ma un semplice algoritmo suddivideva lo schermo in funzione dell'informazione richiesta.

```
<layouts>
  <layout name = "lay1">
    <column>
      <row>
        <monitorable name = "pathname/h1"/>
        <monitorable name = "pathname/h2"/>
      </row>
      <row>
        <monitorable name = "pathname/h3">
      </row>
    </column>
    <column>
      <row>
        <monitorable name = "pathname/h4"/>
      </row>
    </column>
  </layout>
</layouts>
```

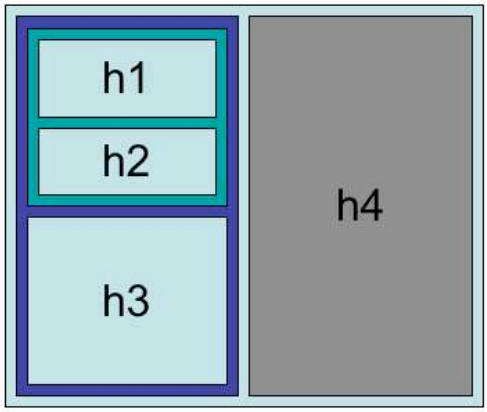

Figura 4.4: Esempio di descrizione XML del layout e relativa visualizzazione.

Da queste limitazioni è nata la necessità di mettere in grado la GUI di sottoscrivere con un solo click tutti i contenuti desiderati e soprattutto di poterli organizzare a proprio piacimento. Il *layout* permette all'utente di richiedere al programma una configurazione predefinita contenente l'informazione da sottoscrivere e visualizzarla come descritto.

Con un layout definiamo quindi quali istogrammi visualizzare e quale disposizione vogliamo che assumano all'interno della finestra di visualizzazione secondo un ordinamento riga/colonna specificabile a piacere.

Senza addentrarsi in dettagli implementativi, i più rilevanti dei quali verrano

descritti nella prossima sezione, è utile spiegare il funzionamento del layout dal punto di vista dell'utente finale. Un layout viene specificato con la creazione di un file XML. Una semplice sintassi permette di specificare appunto gli istogrammi desiderati e come visualizzarli (Figura 4.4). Una volta lanciato il programma principale, questo legge da un'apposita cartella gli eventuali layout presenti e li mostra nella stessa finestra all'interno della quale è rappresentata la struttura ad albero, permettendo quindi all'utente di sottoscrivere un intero layout e quindi tutti i contenuti desiderati.

4.2.1 Implementazione

In questa sezione sono discusse le astrazioni e gli schemi di interrelazione fra componenti utilizzati, facendo riferimento a schemi esemplificativi che mostrano i principali moduli ed i loro collegamenti.

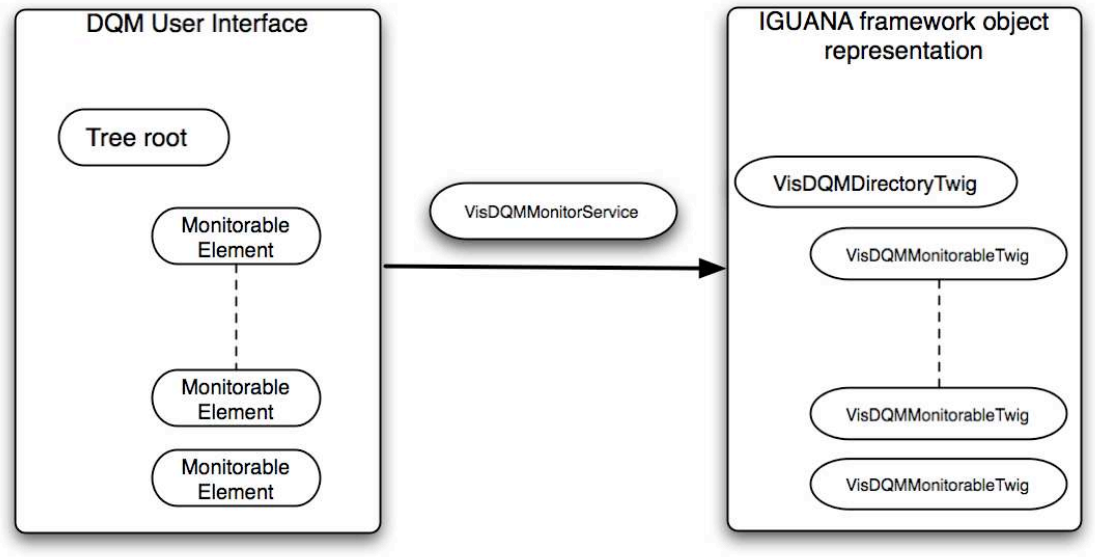

Figura 4.5: Rappresentazione del modello astratto dei dati dell'interfaccia del DQM.

Supponiamo che le informazioni che vogliamo trattare (principalmente visualizzare) siano già disponibili sotto forma di monitorable element e definiamo la *DQM User Interface* (Figura 4.5) come il luogo in cui sono contenute tali informazioni con

le quali siamo interessati ad interagire. Sebbene il termine user interface possa inizialmente far pensare al concetto di interfaccia grafica, la DQM UI rappresenta piuttosto un mezzo attraverso il quale interfacciarsi all'informazione cercata, conservata ed organizzata secondo una struttura ad albero.

Il nostro obiettivo, creare un *modello* astratto che rispecchi la struttura dei dati dell'interfaccia DQM. Il servizio VisDQMMonitorService è il servizio IGUANA che permette di comunicare con la UI e quindi accedere ai monitorable element dei quali andremo a crearci una rappresentazione (il modello astratto appunto) che identifichiamo con gli oggetti VisDQMTwig (*VisDQMDirectoryTwig* e *VisDQMMonitorableTwig*).

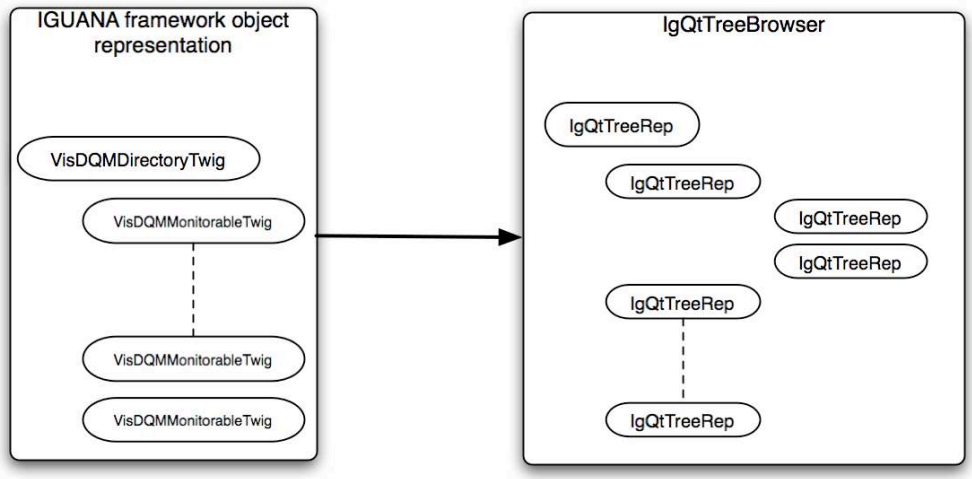

Figura 4.6: Rappresentazione Qt degli oggetti tramite multimetodo.

Questo modo di operare, caratteristico di IGUANA, permette di separare completamente gli oggetti rappresentati dalla loro rappresentazione, che può avvenire utilizzando diverse tecnologie, e il modo di rappresentarli, ossia i *multimetodo* [29]. Nel nostro caso specifico, una prima rappresentazione alla quale siamo interessati avviene attraverso Qt e consiste nella visualizzazione a schermo dell'albero contenente le informazioni, e la possibilità di interagire con esso. L'utilizzo del multimetodo per-

mette il passaggio dai VisDQMTwig astratti alle rappresentazioni *IgQtTreeRep* inserite nel contesto di un IgQtTreeBrowser che ne permette la navigazione (Figura 4.6).

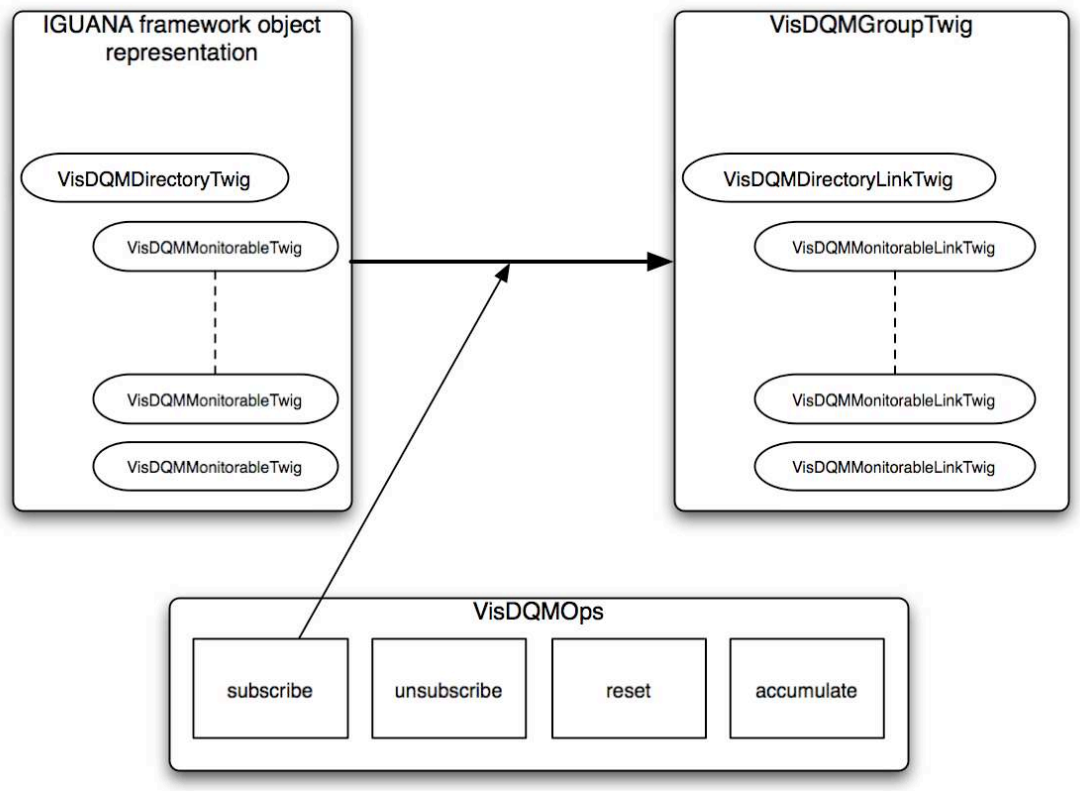

Figura 4.7: VisDQMOps rappresenta il controllo del modello astratto e mappa l'input dell'utente all'interno dell'applicazione.

Alla rappresentazione astratta può poi essere fatta corrispondere una vista *particolare* costituita da link che fanno riferimento al modello. Una tale vista si ottiene a partire dall'uso del controller del modello astratto, *VisDQMOps* in questo caso (Figura 4.7), il quale realizza la mappatura tra input dell'utente e processi eseguiti dal modello, implementando la logica di controllo dell'applicazione. La vista così ottenuta rappresenta a sua volta un modello astratto che fa riferimento a un sottoinsieme dell'informazione complessiva. A questo modello, in quanto tale, possono poi essere associate rappresentazioni ottenibili via multimetodo, ad esempio

si veda (Figura 4.8) la rappresentazione ROOT, risultante dall'applicazione della funzione subscribe da parte del controller.

L'inserimento del concetto di layout è avvenuto in modo analogo a quanto visto in precedenza (Figura 4.7), partendo quindi sempre dal modello astratto ma utilizzando un controller non banale, definito a priori dall'utente con l'inserimento del file xml contenente la descrizione della configurazione desiderata. Dal modello astratto ottenuto in questo modo si può quindi derivare, sempre tramite multimetodo, la rappresentazione Qt, oppure ROOT.

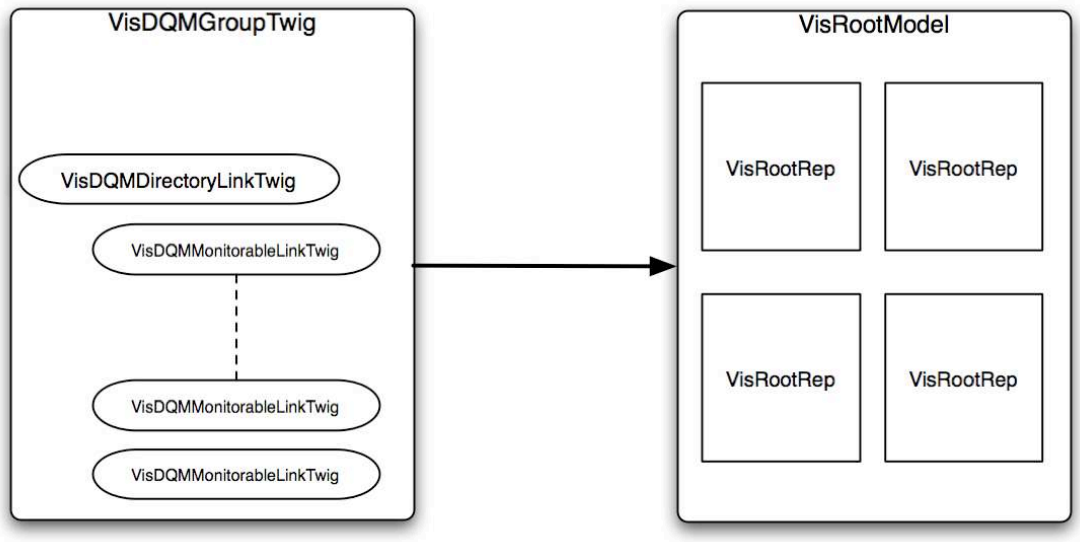

Figura 4.8: Rappresentazione ROOT degli oggetti tramite multimetodo.

L'esempio pratico di quanto appena detto si vede analizzando la finestra grafica di IGUANA (Figura 4.9), dove le due finestre principali altro non sono se non la rappresentazione Qt (a sinistra) e ROOT (a destra) dell'informazione disponibile nella DQM UI, ottenute tramite l'apposito multimetodo.

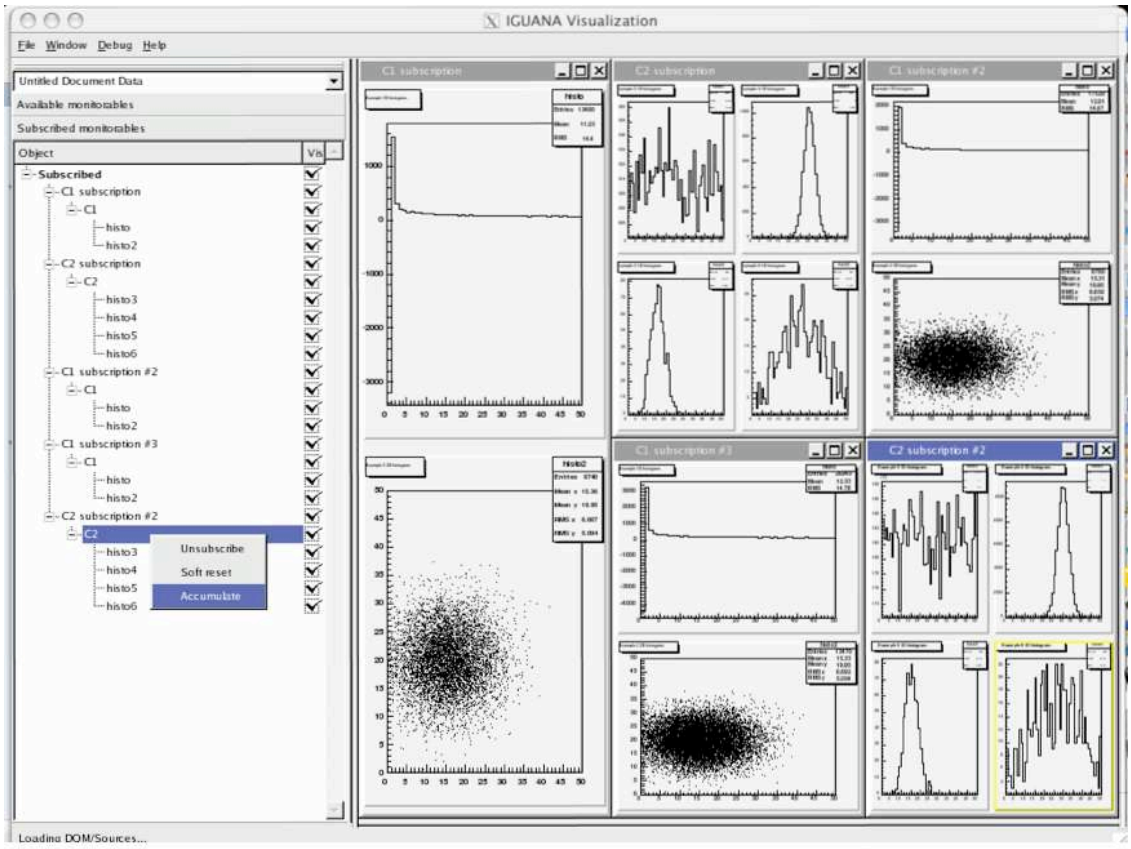

Figura 4.9: Finestra di visualizzazione di IGUANA: sulla sinistra la rappresentazione Qt ad albero dei dati. Sulla destra la rappresentazione grafica ottenuta con ROOT, i dati rappresentati sono i medesimi ma cambia il modo di rappresentarli.

4.3 Analisi e monitoring del DQM

Una volta concluso lo sviluppo del plugin appena descritto è stata fatta la scelta, come già detto in precedenza, di analizzare performance e robustezza del sistema a basso livello, con lo scopo di identificare punti deboli e *bottlenecks*. L'idea iniziale è stata creare, in un certo senso, un sistema di monitoring del sistema di monitoring, ossia un sistema in grado di controllare lo stato del framework DQM; nello specifico eventuali congestioni nelle comunicazioni fra source, client e collector, in caso di stress del sistema.

Vediamo adesso più in dettaglio la modalità di comunicazione fra source e collector: questo modo di procedere è stato scelto per avere un modello da studiare prima di passare alla fase di implementazione, così da poter formulare alcune ipotesi sul funzionamento del sistema per poi decidere quali test implementare e come. Supponiamo, per semplicità, di studiare una situazione in cui le componenti in gioco consistono solamente in una source, un collector ed un client, che la source abbia già pubblicato al collector la lista di istogrammi disponibili organizzati per directory e che il client abbia eseguito una richiesta di sottoscrizione per tutta l'informazione disponibile. Quest'ultima consiste in $h + k$ istogrammi distribuiti in due directory, h nella prima e k nella seconda.

Vediamo adesso lo schema di funzionamento prima della generica source e successivamente quello del collector, tenendo a mente le ipotesi di cui sopra.

Source

Una volta che la source ha ricevuto una richiesta per tutta l'informazione in suo possesso (gli $h + k$ istogrammi) inizia ad effettuare i cosiddetti cicli di monitoring: una prima fase, che viene ripetuta per tutte le directory sottoscritte, consiste nel

riempire il messaggio corrente con i dati da inviare, successivamente inviare una stringa contenente il pathname della directory corrente concatenato al numero di istogrammi in essa contenuti ed infine inviare i dati effettivi. Una volta eseguita questa prima fase su tutte le directory sottoscritte, la source invia al collector un ulteriore messaggio avvisandolo della fine del ciclo di monitoring (Figura 4.10).

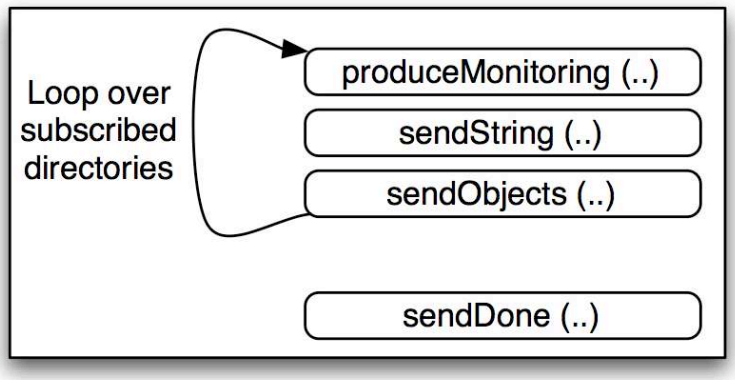

Figura 4.10: Procedura di invio dell'informazione di monitoring da parte della source: produceMonitoring (..) riempie il messaggio corrente con i dati da inviare, sendString (..) invia una stringa del tipo "pathname+numero di oggetti da spedire mentre sendObjects (..) invia i dati effettivi. Tutto questo viene iterato sul numero di directory che sono state sottoscritte, dopodichè viene eseguito sendDone (..) per inviare un messaggio che indica la fine del ciclo di monitoring.

Collector

Il collector riceve messaggi in ingresso e, prima di inoltrare gli istogrammi al client, attende che la source abbia terminato un ciclo di monitoring, ossia attende la ricezione del messaggio che segnala la fine del ciclo. Ciò vuol dire che, sebbene il collector abbia già ricevuto dell'informazione, esso attende comunque la fine di un ciclo prima di poterla inviare al client che ne ha fatto richiesta (Figura 4.11). Questo avviene poichè il collector implementa il caching degli istogrammi che riceve, in modo da poter soddisfare eventuali richieste da parte di altri client rispetto a una stessa informazione, senza dover attendere il successivo ciclo di monitoring. I messaggi di

risposta vengono creati a partire dal contenuto della cache, e le richieste soddisfatte tramite l'invio di un unico messaggio contenente tutta l'informazione richiesta.

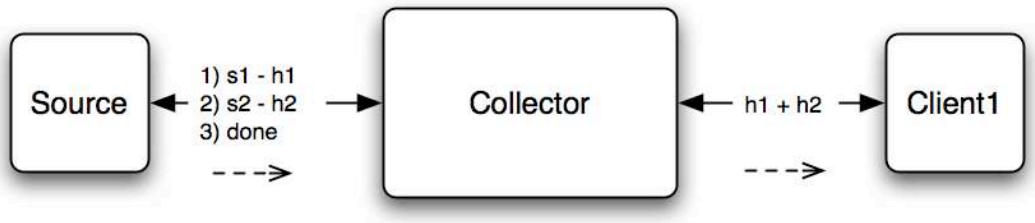

Figura 4.11: Invio informazione sottoscritta: il collector riceve cinque messaggi, le prime due coppie contegono le informazioni relative alle rispettive directory sottoscritte, il terzo è la segnalazione di fine ciclo. A questo punto il collector spacchetta le informazioni per eseguire il caching e successivamente invia al client l'informazione richiesta ricompattata in un unico messaggio.

Questa analisi mi ha portato ad avere alcuni dubbi sia sulla robustezza che sull'effettiva efficienza del sistema; nello specifico le questioni principali da me sollevate sono state due:

- Nell'esempio sopra il collector interagisce con una sola source, l'algoritmo presentato garantisce robustezza nel caso la comunicazione veda coinvolga più di una source?
- Per quale motivo è necessario che il collector elabori l'informazione ricevuta invece di smistarla semplicemente ai client?

Lasciando la risposta alla seconda domanda al capitolo successivo, in quanto legata più ad un aspetto teorico e a scelte di implementazione, andiamo ad analizzare invece le problematiche legate al primo punto, in quanto, come vedremo fra breve, esse possono portare ad effettivi malfunzionamenti del sistema.

4.3.1 Interazioni collector-source

Supponiamo di avere un sistema con n source e m client (Figura 4.12) e che la fase di sottoscrizione di tutti i componenti sia già avvenuta, così che le uniche informazioni che circolano nel sistema sono istogrammi. Il collector in generale segue uno schema di funzionamento del tipo:

- loop sulle source per ricevere nuovi istogrammi;
- elaborazione dei messaggi da inviare ai client;
- loop sui client per inoltrare gli istogrammi richiesti.

Figura 4.12: Schema di funzionamento del collector nel caso sia connesso a più source e client.

Analizzando il codice sorgente si scopre che il primo loop viene chiuso solo una volta che tutti i cicli di monitoring iniziati si siano conclusi. Questo fatto non comporta alcun problema nel caso ci sia una sola source, ma in caso contrario può portare a un *deadlock* causato da una serie continua di rimbalzi.

Se si verifica infatti la situazione:

- $S1$ invia istogrammi ma ancora non ha inviato il messaggio di *done*;
- $S2$ invia istogrammi ma ancora non ha inviato il messaggio di done;
- $S1$ invia il messaggio di done;
- $S1$ invia istogrammi in un nuovo ciclo ...

Il collector non esce dal loop ma attende la chiusura dei cicli di monitoring sospesi (nell'esempio il done da parte di $S2$) prima di iniziare l'elaborazione interna e la successiva fase di spedizione ai client. Tuttavia, durante l'attesa del done da S2, S1 inizia un nuovo ciclo di update.

È facile immaginare come, al crescere del numero di source e in funzione della frequenza di interpartenza (prefissata per ogni source) possa venire a crearsi un loop di continui rimbalzi che impediscono al collector di passare al loop sui client, causando il blocco di questi ultimi, in attesa di informazioni che non arriveranno mai. L'analisi dei risultati ha inoltre permesso di individuare un'altra situazione di instabilità che non porta solamente al blocco dei client ma anche a quello della source (nel caso precedente queste potevano essere solo rallentate) e, in quanto legata ad un problema nella sincronizzazione dei messaggi durante la fase di sottoscrizione, è indipendente dal loro numero.

La volontà di avere un sistema robusto ed adattabile all'evolversi delle esigenze nel tempo, ha fatto sì che l'interesse del gruppo, e quindi il mio lavoro, si spostasse dall'intento iniziale di creare un sistema di monitoring al verificare se effettivamente le soluzioni ipotizzate trovassero riscontro reale.

Allo scopo iniziale di accertare la presenza dei problemi visti sopra si è quindi aggiunto quello di identificare potenziali soluzioni.

4.3.2 Pianificazione

Poichè il collector rappresenta l'unico componente modificabile, e in un certo senso il più critico, di questa architettura, il lavoro di implementazione si è concentrato su di esso. Il canale di comunicazione tra insieme di source e collector è asimmetrico in ingresso a quest'ultimo con grado di asincronia del canale pari a uno ed ognuna delle n source invia dati con una frequenza (tempo di interpartenza dalla source) fissata a priori. Si conclude quindi che se il tempo che occorre al collector per servire due richieste consecutive provenienti dalla stessa source è inferiore o uguale al tempo di interpartenza visto poco fa, il sistema lavora come dovrebbe, anche se questo non esclude assolutamente la presenza di una situazione di deadlock. Indicando con ΔS_i l'intervallo di tempo impiegato dal collector per ricevere due messaggi dalla i-esima source e con Δt la frequenza prefissata di interpartenza dalla source, sarà preferibile il verificarsi della disequazione $\Delta S_i \leq \Delta t$. È però opportuno far presente che all'aumentare del numero delle source contemporaneamente impegnate in comunicazioni, risulta inevitabile che superato un certo n tale situazione non sará comunque verificabile.

Quello che si è cercato di fare è stato di fornire un modo per misurare il valore ΔS_i relativo alla source i, essendo noti Δt, n e la dimensione del messaggio che attraversa il link. Per ciascuna source viene creato un file contenente tutte le informazioni riguardanti ogni effettivo invio di dati, file che permette lo studio a posteriori di come cambiano le prestazioni del sistema al variare della frequenza di interpartenza, del numero di source e della dimensione del messaggio. L'obiettivo è quello di verificare l'omogeneità dei dati raccolti per tutte le source accertando che non si verifichino situazioni di squilibrio di carico che potrebbero portare a deadlock.

4.3.3 Implementazione

La concretizzazione di quanto detto in precedenza è avvenuta con l'implementazione di una classe chiamata *TimeMonitor*, facente parte della nuova libreria *Diagnostic* del framework, destinata a contenere tutte le funzioni di utilità per il monitoring di sistema. L'idea di base è quella di andare a misurare i tempi ΔS_i in modo indiretto, ossia misurando il tempo che impiega il collector a ricevere due informazioni consecutive da una stessa source S_i. Nel caso si decidesse di avviare il collector con il sistema di diagnostica attivo, il costruttore di quest'ultimo si preoccuperà di creare un'istanza della classe TimeMonitor.

Nell'interfaccia di TimeMonitor è definita la *struct*:

```
struct TimeData
{
   seal::TimeInfo::NanoSecs doneTime;
   seal::TimeInfo::NanoSecs startTime;
   std::ofstream file;
   int count;
   int iterCounter;
   int messSize;
};
```

Il campo fondamentale di tale struttura è quello che rappresenta il file, *std::ofstream file*, che andrà a contenere gli intervalli di tempo misurati, mentre gli altri sono variabili di utilità di cui *doneTime* e *startTime* facenti parte della libreria SEAL [30] *TimeInfo* utilizzata per l'implementazione dei cronometri.

La struct vista sopra è poi usata per definire una mappa

```
std::map <const char*, TimeData*> timeMap;
```

che viene utilizzata per associare univocamente ad ogni TimeData una determinata source (*const char**), e quindi il file (*std::ofstream file;*) contenente le informazioni desiderate (Figura 4.13).

```
ADDING: FU02 at time: 1.98345e+15
1   Sorgente: FU02   DeltaTime: 1.6e+08
  SIZE: 376299
2   Sorgente: FU02   DeltaTime: 1.13e+09
   SIZE: 376299
3   Sorgente: FU02   DeltaTime: 1.18e+09
   SIZE: 376299
4   Sorgente: FU02   DeltaTime: 1.18e+09
   SIZE: 376299
5   Sorgente: FU02   DeltaTime: 1.19e+09
   SIZE: 376299
6   Sorgente: FU02   DeltaTime: 1.22e+09
   SIZE: 376299
7   Sorgente: FU02   DeltaTime: 1.17e+09
   SIZE: 376299
8   Sorgente: FU02   DeltaTime: 1.17e+09
   SIZE: 376299
9   Sorgente: FU02   DeltaTime: 1.19e+09
   SIZE: 376299
```

Figura 4.13: Esempio: contenuto del file associato alla source FU02.

4.3.4 Considerazioni

L'analisi dei dati raccolti ha messo in evidenza alcuni fatti interessanti ma non ha dato purtroppo segnali incoraggianti sull'effettiva robustezza del sistema. Se infatti con un numero di source limitato ($n < 8$) il sistema si comportava in modo regolare, con valori di n superiori le oscillazioni nei valori misurati erano tali da far pensare a qualche errore dovuto a dettagli di programmazione.

Ulteriori analisi hanno infatti messo in evidenza un'implementazione nondeterministica della funzione *select()* sul socket in ingresso al collector. Questo vuol dire che, potenzialmente, alcune source potevano venire ignorate completamente, fatto che oltre a spiegare le oscillazioni misurate è stato un campanello di allarme in quanto tali oscillazioni rappresentavano tempo durante il quale la source era bloc-

cata su una send [1], cosa non ammissibile, vista la criticità dei processi in esecuzione su di essa. Il proseguire dei test sul sistema ha poi messo alla luce un problema di sincronizzazione fra source, collector e client durante lo scambio di richieste di sottoscrizione per un volume di dati superiore a 50000 istogrammi, in tali casi infatti la richiesta viene spezzettata in più richieste, comportamento che evidentemente non era stato previsto e che causa il blocco delle sorgenti.

Il presentarsi di tutti i problemi qui analizzati ha concretizzato i dubbi che riguardavano l'architettura con collector, la decisione del gruppo è stata così quella di continuare il supporto a tale implementazione, apportando miglioramenti dove possibile, e nel frattempo indagare nuove possibli soluzioni. Sebbene il passaggio allo storage manager rispondesse ai problemi più critici, ossia relativi alla non interrompibilità dei processi in esecuzione sulle FU, per quanto riguarda la visualizzazione invece il problema rimane aperto. Fino ad ora il funzionamento classico ha sempre visto l'interfaccia grafica (applicazione client) ricevere istogrammi tramite collector, soluzione che abbiamo visto però essere inaffidabile e quindi da risolvere in modi differenti.

4.4 Ottimizzazione del framework

Due possibili design che evitano il problema appena discusso, sono esposti nel capitolo seguente. Completiamo invece nel seguito l'esposizione delle modifiche comunque apportate all'architettura con collector fino alla fine del suo periodo di vita ed esaminiamo alcune componenti di ROOT il cui utilizzo era problematico e che sono state sostituite da altre. Le modifiche qui descritte saranno sostanzialmente due: la prima mirata a definire un tipo unico per i messaggi che circolano nel framework,

[1] Ricordiamo che il grado di asincronia del canale fra source e collector è pari ad uno, quindi al secondo invio, se il primo messaggio non è stato ricevuto, il mittente si blocca.

in modo da avere un formato chiaro e flessibile per identificare l'informazione; la seconda riguardante la riscrittura completa delle comunicazioni fra processi a livello di trasporto, in modo da ottimizzare, dove possibile, la gestione dei socket, ed eliminando parte dei problemi visti in precedenza.

ROOT

Il framework ROOT è largamente utilizzato all'interno di CMSSW per quanto concerne l'analisi dei dati. Esso fornisce funzioni di utilità atte a risolvere tutte le problematiche relative all'implementazione di strutture di dati complesse, alla loro analisi, e al loro trasporto ed è stato utilizzato pesantemente per la scrittura del codice del DQM. Sebbene molto indicato ed apprezzato per la generazione di plot ed istogrammi, non è ben visto da una buona parte degli sviluppatori a causa della presenza di un numero elevato di variabili statiche sparse per il codice che creano confusione e, a volte, causano comportamenti inattesi delle applicazioni.

Le parti di tale framework che interessano la mia trattazione riguardano la classe *TMessage*, che definisce il messaggio che incapsula le informazioni, la classe *TBuffer*, che definisce il buffer contenitore dei dati serializzati ed infine la classe *TSocket*, utilizzata per la gestione delle comunicazioni.

4.4.1 Formato messaggi

In vista delle successive modifiche da apportare al layer di trasporto è stata creata una classe apposita, *DQMMessage*, che definisce il formato dei messaggi circolanti nel framework, evitando così di essere legati alla struttura imposta da *TMessage* in eventuali futuri aggiornamenti. I dati effettivi continuano ad essere incapsulati in un oggetto di tipo *TBuffer* e, come in precedenza, sono presenti due variabili

che tengono conto del tipo di messaggio incapsulato (*m_what*) e della sua lunghezza (*m_length*).

```
DQMMessage (TBuffer* buffer, unsigned int what);
TBuffer *           m_buffer;
unsigned int        m_what;
unsigned int        m_length;
```

L'oggetto DQMMessage ha una struttura estremamente semplice, legata alle necessità del framework DQM e soprattutto, in quanto specializzata, non soggetta a modifiche inattese da parte di terzi, che potrebbero pregiudicare il funzionamento del sistema.

4.4.2 Modifiche a livello di trasporto

Le comunicazioni fra processi del DQM basate su ROOT, quindi su TSocket, fanno uso, a livello di trasporto, del protocollo TCP. Il nuovo sistema di trasporto dei dati utilizza pure TCP, ma permette maggiore flessibilità e controllo sui socket, in modo da avere una base più stabile in vista dei futuri cambiamenti. La libreria implementata in ROOT per la gestione delle comunicazioni di rete è molto funzionale ma, nel caso in esame, non permette un completo controllo dei dati che attraversano i link, e quindi non rappresenta la soluzione ideale.

La scelta iniziale è caduta sulla libreria *SealSocket* che però, dopo diversi tentativi di implementazione e svariati test, si è rivelata essere inadeguata agli scopi. La scelta è infine stata quella di utilizzare direttamente i socket *posix*, ossia quelli standard per sistemi operativi *UNIX*, e pertanto utilizzati a basso livello anche da ROOT. Sebbene questa scelta abbia causato un overhead, in termini di tempo di sviluppo, tutt'altro che irrilevante rispetto all'utilizzo delle classi disponibili in

ROOT, essa ha permesso il controllo totale delle comunicazioni, che si è tradotto in un utilizzo più efficiente delle risorse.

Implementazione socket

Mostriamo ora le principali differenze fra le due metodologie, vediamo prima le differenze nella creazione di un socket lato server *ServerSocket* (il socket per le comunicazioni in uscita ha una forma analoga) e successivamente come è stata implementata la funzione *fair Select()*, utile a garantire equità nella risposta da collector a source.

La creazione di un ServerSocket in ROOT è molto semplice:

ROOT infatti definisce un oggetto *TServerSocket*, la cui istanziazione necessita come parametri esclusivamente del numero di porta *port* e del boolean *kTRUE*, utile a settare il riutilizzo del socket. Una volta creato tale oggetto è possibile poi accettare connessioni invocando il metodo:

```
TSocket* Accept (UChar_t Opt = 0)
```

Tale metodo restituisce un puntatore a un oggetto *TSocket* che utilizzeremo per comunicare col partner remoto utilizzando i metodi

```
virtual Int_t Send(const TMessage& mess);
virtual Int_t Recv(TMessage*& mess);
```

Essi accettano come parametri direttamente oggetti di tipo TMessage e quindi nascondono all'utente tutta la fase di bufferizzazione e spedizione sul canale che invece, in una implementazione ad-hoc, dovremo gestire "a mano".

Il codice per dichiarare un socket lato server in posix è invece del tipo:

```
int ss = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
 sockaddr_in *s_address = new sockaddr_in ();
s_address->sin_addr.s_addr = INADDR_ANY;
s_address->sin_port = htons (port);
s_address->sin_family=AF_INET;
int status = bind (ss, (sockaddr*) s_address, sizeof (sockaddr_in));
listen (ss,5);
```

La prima grande differenza con il caso precedente è che adesso non esiste più il concetto di un oggetto *Socket*: esiste un'astrazione che identifica quest'ultimo con un intero ad esso associato, il descrittore del socket (*int ss*). La creazione del socket necessita dei parametri:

- dominio internet (Unix o IP) [31]
  La macro *AF_INET* indica l'utilizzo del dominio IP;
- tipo di socket (Datagram, Byte-Stream o Sequenced-packet)
  La macro *SOCK_STREAM* indica l'utilizzo dei socket Byte-stream;
- protocollo IP utilizzato (TCP o UDP)
  La macro *IPPROTO_TCP* indica l'utilizzo del protocollo TCP;

Le righe successive consistono nella creazione di un indirizzo (definito tramite la struct *sockaddr *address*) da associare al socket. Tale associazione, detta in termini tecnici *binding*, viene effettuata invocando la funzione:

```
int bind (int socket, const struct sockaddr *address, socklen_t address_len);
```

in quanto i socket creati con la funzione *socket()* sono inizialmente *unnamed*.

Infine viene chiamata la funzione:

```
int listen (int socket, int backlog);
```

che abilita il socket passato come argomento ad accettare connessioni.

Implementazione funzione select()

In ROOT la classe TServerSocket include il metodo:

```
TSocket* Accept (UChar_t Opt = 0);
```

L'invocazione di tale metodo da parte di un'istanza di TServerSocket, indica che tale socket accetta connessioni. Una volta avvenuta una connessione, esso restituisce un puntatore al TSocket che verrà utilizzato per la comunicazione.

Facendo riferimento al problema relativo alla comunicazione fra n source ed un collector, è necessario che quest'ultimo abbia un modo di conservare le connessioni aperte in quanto, una volta che una source si connette, tale connessione rimane attiva fino a che ci sono dati da spedire.

Per ovviare a questo problema ROOT utilizza un oggetto di tipo *TMonitor*, esso include una serie di metodi per l'arbitraggio di un insieme di socket. Una volta creata un'istanza di tale classe è possibile invocare i metodi:

```
virtual void Add(TSocket* sock, Int_t interest = kRead);
TSocket* Select ();
```

Il primo aggiunge *sock* alla lista dei monitor attivi, mentre il secondo restituisce uno fra quelli che dispongono di dati in attesa di essere ricevuti.

Per quanto riguarda invece i socket posix, esistono una funzione (*accept*) che restituisce il descrittore del socket da utilizzare per la comunicazione ed una funzione (*select*) che modifica una maschera di bit, settando quelli relativi ai descrittori che dispongono di dati in lettura:

```
int accept (int socket, struct sockaddr *address, socklen_t *address_len);
int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
struct timeval *timeout);
```

Tralasciando la descrizione della prima funzione in quanto intuitiva, della seconda gli unici campi di interesse sono i primi due e l'ultimo:

- */int nfds/* specifica l'intervallo di descrittori da testare, la funzione controlla tutti quelli compresi fra 0 ed nfds-1
- */fd_set *readfds/* rappresenta la maschera che tiene conto dei descrittori disponibili in lettura. I due argomenti successivi sono analoghi a questo e rappresentano maschere di bit, rispettivamente, per scrittura e per test di errori
- */struct timeval *timeout/* specifica l'intervallo di tempo da attendere prima di completare la selezione

Sfruttando la maschera di bit (*rmask* nel codice che segue), è stato quindi appositamente sviluppato un algoritmo che agisce sulla selezione dei socket in lettura (Figura 4.14). Mentre nella soluzione precedente tale operazione non dava alcun tipo di importanza ad una qualsiasi forma di ordinamento o priorità, permettendo potenzialmente più letture consecutive su di uno stesso socket, e quindi portando ai problemi visti nello scorso capitolo, la soluzione qui presentata invece garantisce prestazioni e distribuzione del carico equilibrate.

Per le modifiche alla maschera di bit vengono utilizzate le macro FD_CLR(fd, &fdset), FD_SET(fd, &fdset) e FD_ZERO(fd, &fdset) [2], il funzionamento dell'algoritmo è molto semplice e può essere riassunto nei seguenti punti:

- viene eseguita la *select* nel caso non sia già stato fatto in precedenza;
- all'esecuzione della select, la maschera rmask risulterà settata per tutti i descrittori attualmente disponibili in lettura;
- si esegue un loop su tutti i descrittori, il primo che troviamo settato nella maschera di bit viene resettato ed il socket ad esso corrispondente settato come socket corrente e letto;
- nelle successive iterazioni la select non viene mai chiamata fino a che esistono descrittori settati.

```
if (!selectCalled)
   {
     selectCalled = true;
     selectSock();
   }
//now rmask is set with all the active sockets, just loop over all of them
  for (int fd = 0; fd <= monMax; ++fd)
   {
     if (fd == monMax) selectCalled = false;
     if (FD_ISSET (fd, &rmask))
      {
        FD_CLR (fd, &rmask);
        currs = fd;
        string buffer;
        Int_t msize = SocketUtils::readMessage (mess, currs);

        (.....)
      }
   }
  return false;
}
```

Figura 4.14: Estratto dell'algoritmo che agisce sulla selezione dei socket in lettura.

[2]FD_CLR (fd, &fdset) resetta il bit corrispondente al descrittore fd nella maschera fdset, FD_SET (fd, &fdset) lo setta, mentre FD_ZERO (fd, &fdset) resetta l'intera maschera.

Implementazione lettura/scrittura socket

La libreria TSocket mette a disposizone le due funzioni:

```
virtual Int_t Send(const TMessage& mess);
virtual Int_t Recv(TMessage*& mess);
```

Queste risultavano essere molto comode in quanto operano direttamente su un oggetto TMessage. Per ottenere lo stesso tipo di funzionalità è stata creata una libreria si supporto, *SocketUtils*, che, mascherando all'utente le problematiche relative alla bufferizzazione dei messaggi sulla rete, fornisce un'interfaccia semplice ed analoga alla precedente:

```
int sendMessage (DQMMessage *mess , int sock_des);
int sendString (const char *mess, int sock_des);
int readMessage (DQMMessage*& mess, int sock_des);
```

4.5 Considerazioni

Nel capitolo in esame sono state trattate tutte le modifiche da me applicate all'infrastruttura del DQM che sono state approvate ed inserite in modo stabile nel codice del framework effettivamente utilizzato nell'esperimento CMS. Si è deciso di non fare riferimento alle implementazioni dei test ed al codice di supporto e mantenimento, sia in riferimento al mio lavoro in particolare, sia per quanto riguarda altre problematiche, emerse o all'interno del gruppo stesso o da collaborazioni con altri gruppi, poichè, in quanto riferite ad altri contesti non sono state ritenute pertinenti agli argomenti qui trattati.

La tecnologia layout è stata utilizzata da tutti i sottogruppi per lungo tempo, fino a quando l'interfaccia DQM web non ha raggiunto uno stadio avanzato di svi-

luppo ed ha sostituito completamente la vecchia GUI. Le modifiche da me effettuate all'infrastruttura delle comunicazioni sono tutt'ora utilizzate e tutti i problemi emersi durante la fase di analisi e di sviluppo hanno contribuito largamente ad accertare la necessità di operare cambiamenti più o meno radicali all'interno del framework per garantire un sistema efficiente in vista dell'attivazione dell'esperimento.

Capitolo 5

Sviluppi futuri e conclusioni

5.1 Introduzione

In questo capitolo vengono discusse le proposte di modifica all'architettura del DQM con Collector. Due soluzioni sono state studiate, una basata sul mantenimento del ruolo del collector, l'altra basata invece sull'utilizzo di un server web. Quest'ultima è stata infine adottata con successo dall'interfaccia IGUANA DQM web.

Entrambe le achitetture hanno i seguenti punti comuni:

- utilizzano un nodo intermediario fra source e client;
- il nodo intermediario è un semplice router delle informazioni e non esegue alcuna operazione su di esse o ne analizza il contenuto.

5.2 Nuova Architettura del Collector

Il collector è un elemento troppo complesso e carico di funzionalità. In questa trattazione si intende sia semplificare la struttura del componente in esame, sia

ridurre il numero di comunicazioni e rimbalzi fra di esso ed i componenti ad esso connessi, ossia le varie source e i client.

Supponiamo di trovarci nella situazione in cui due client fanno richiesta di sottoscrizione per due insiemi di istogrammi in parte coincidenti e che sia presente una sola source (Figura 5.1).

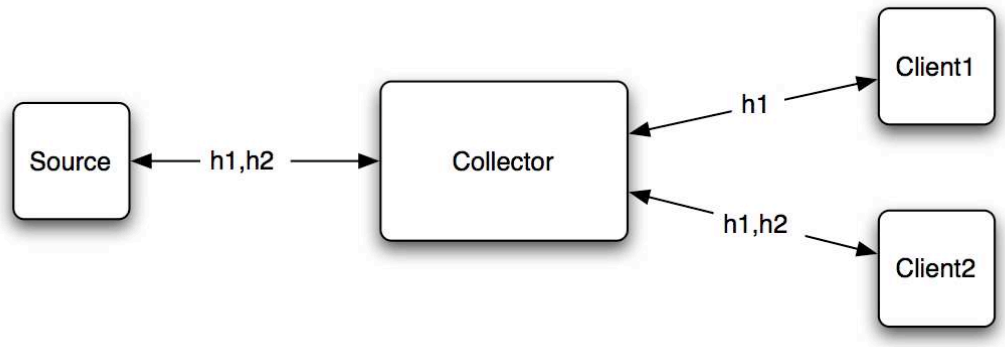

Figura 5.1: I due client fanno richiesta di sottoscrizione di informazioni in parte coincidenti.

Il comportamento del collector a questo punto è quello di popolare una mappa (MAP1, Figura 5.2), associando a ciascun istogramma la lista di client che ne hanno fatto richiesta.

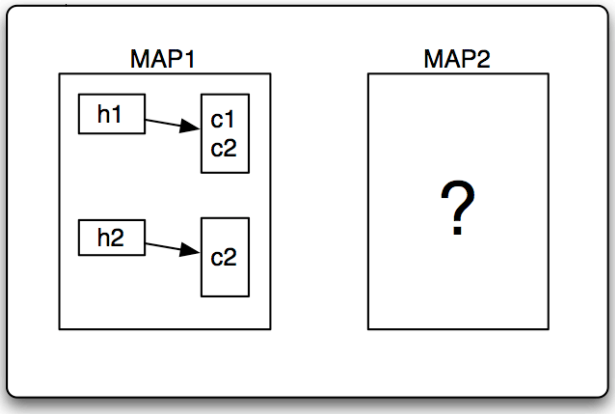

Figura 5.2: Una volta ricevuta una richiesta di sottoscrizione il collector popola MAP1, MAP2 sarà esaminata in seguito.

Successivamente il collector spedisce alle source corrispondenti le richieste di sottoscrizione. A questo punto sono necessarie alcune modifiche negli algoritmi che gestiscono l'invio (lato source) e ricezione (lato collector) dei dati contenenti gli istogrammi. Ricordiamo che il ciclo in questione consiste in quattro fasi, le prime tre delle quali iterate sul numero di directory sottoscritte:

- produrre l'informazione;
- inviare una stringa contente il pathname concatenato al numero degli oggetti spediti;
- inviare i dati corrispondenti agli oggetti;
- inviare il messaggio di done.

La ridondanza generata dall'invio di questi messaggi è stata causa della maggior parte dei malfunzionamenti riscontrati. La soluzione qui proposta consiste nel semplificare il tutto, sostituendo i quattro step con due semplici passi (Figura 5.3): al primo corrisponde la produzione dell'informazione mentre al secondo il suo inoltro sulla rete. L'unica ulteriore modifica riguarda il primo dei precedenti due punti, cosa che rende necessario modificare anche la funzione *produceMonitoring(..)*.

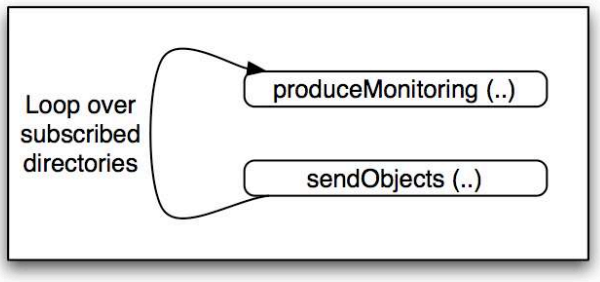

Figura 5.3: Nuovo schema di funzionamento dell'algoritmo di send della source.

Tale funzione, che in precedenza si occupava solo di riempire un buffer con i dati per creare il messaggio DQM, adesso deve anche inserire un'intestazione che identifichi il contenuto del messaggio e riferisca la sua posizione all'interno di esso,

così facendo non sarà più necessario che il collector elabori ogni volta il messaggio richiesto. Per chiarire il comportamento del collector nella fase di ricezione lato source vediamo ora che funzione ha il componente MAP2 (Figura 5.4).

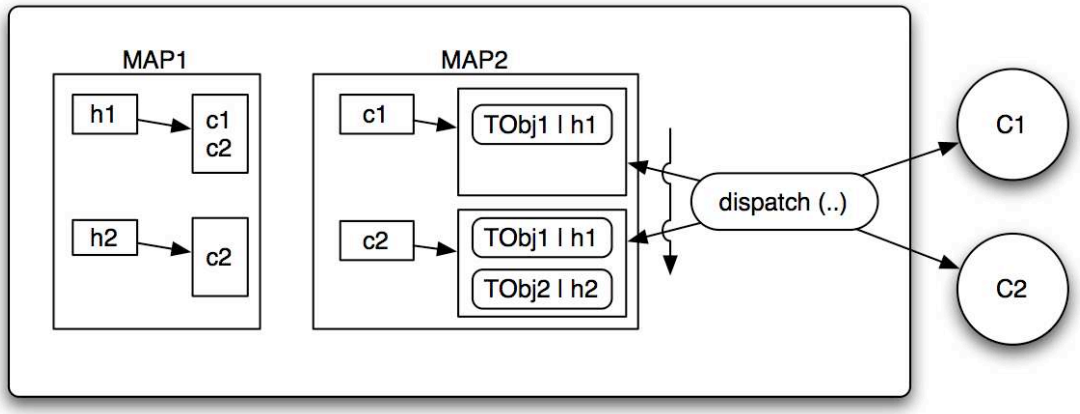

Figura 5.4: Funzionamento del collector durante il passaggio di informazioni da source a client.

Non appena il collector riceve un messaggio ne esamina l'intestazione e, istogramma per istogramma, identifica utilizzando MAP1 quali client ne hanno fatto richiesta. A questo punto, utilizzando solo puntatori al contenuto del messaggio ricevuto, popola la mappa MAP2 associando ad ogni client l'informazione da lui richiesta. Infine una funzione *dispatch (..)* in uscita dal collector si occupa, rispettando un delay impostato a programma o predefinito, di inoltrare gli istogrammi ai rispettivi client.

5.3 Architettura basata su Web Server

Questa seconda soluzione è stata pensata nell'ottica del crescente interesse mostrato dai gruppi verso l'interazione con i dati via browser web. Essa si basa sull'utilizzo di un server web standard, che si occupa di veicolare le richieste e i dati dagli utenti ai processi DQM e viceversa. Non interessa ai fini della trattazione qui esposta

il modo in cui l'utente finale interagisce con i dati, quindi potrà essere prevista sia un'interfaccia standard eventualmente basata su IGUANA, sia l'utilizzo di un browser web, soluzione comunque ideale viste le tecnologie utilizzate.

Le richieste di sottoscrizione e in generale tutti i dati che riguardano le comunicazioni fra utente e server web sono incapsulati adesso in richieste HTTP. In seguito alla ricezione dei dati il server web segnala ai processi DQM [1] in esecuzione su un nodo separato, i dati per i quali è stata fatta richiesta di sottoscrizione (Figura 5.5).

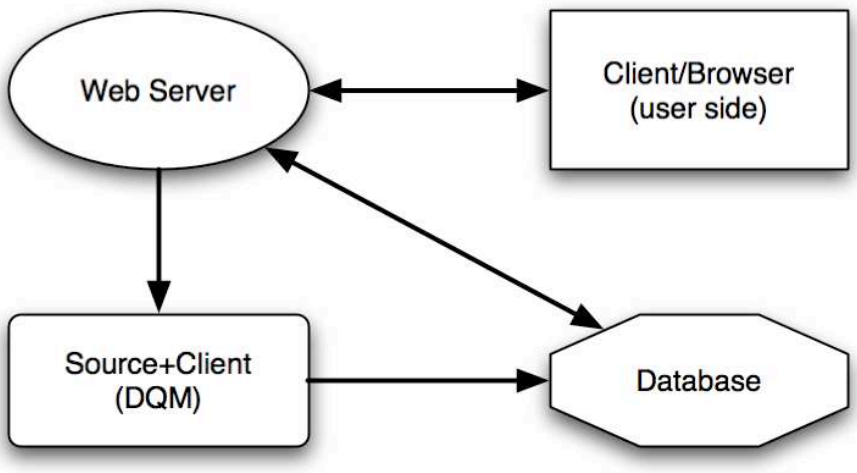

Figura 5.5: Schema esemplificativo delle connessioni fra componenti nell'architettura con web server.

Il protocollo da usare per questa seconda fase delle comunicazioni non è specificato ed è lasciato al programmatore quale decisione di implementazione.

A questo punto sul nodo di esecuzione viene preparata l'informazione richiesta, che viene memorizzata su database. Sarà compito di quest'ultimo inviare il messaggio di risposta al server web, il quale infine lo recapiterà all'utente. Il cana-

[1] Con processi DQM si intende l'esecuzione simultanea su di uno stesso nodo di source e client, funzionalità resa possibile dal momento in cui è stato abbandonato l'uso del collector.

le di comunicazione in uscita dal server verso il database serve nel caso si desideri accedere a dati relativi a run trascorsi oppure a file ROOT locali.

Concludendo si può inoltre aggiungere che il vantaggio di un tale approccio consiste nel fatto che, con l'eventuale inserimento nel sistema di un server proxy, diventerebbe possibile demandare a quest'ultimo l'accesso autenticato al server web, in modo da rendere la visualizzazione delle informazioni esattamente la stessa, sia in caso di accesso locale, che in caso di accesso remoto.

5.4 Conclusioni finali

Con i due studi qui presentati si conclude il mio lavoro di tesi, lavoro che ha rappresentato un'esperienza fondamentale per la mia crescita professionale. Il far parte di un gruppo di professionisti in un ambiente internazionale è stato, fra le altre cose, di grande aiuto nel migliorare le mie doti comunicative e la capacità di lavorare in team su un progetto distribuito su larga scala.

Il lungo periodo di permanenza al CERN ha inoltre permesso alla trattazione di andare a toccare in modo concreto praticamente tutte le tematiche che hanno interessato il lavoro qui descritto, anche quelle meno rilevanti ai fini di quest'ultimo e quindi non trattate in questa sede. Molto tempo è stato dedicato allo studio del contesto in cui le problematiche da me affrontate erano collocate, e ne abbiamo visto la descrizione nei primi due capitoli. L'inizio dell'implementazione vera e propria ha poi portato ad una specializzazione nei confronti degli specifici argomenti di interesse, ma, essendo state toccate quasi tutte le componenti del framework, dalla visualizzazione alle comunicazioni fra processi, si può affermare che lo studio effettuato è stato completo ed approfondito in tutte le sue parti.

Anche una volta terminato il mio contratto, ed in parallelo agli impegni uni-

versitari, ho continuato a collaborare con il gruppo del tracciatore dell'INFN di Pisa allo sviluppo software per il DQM, e figuro tuttora come *external CMS collaborator* presso il CERN.

Ringraziamenti

Ringrazio di cuore la mia famiglia perchè non mi ha fatto mai mancare il suo appoggio, nemmeno nei momenti più difficili.

Ringrazio il Dott. Emilio Meschi, la cui preziosa guida e il costante incoraggiamento mi hanno permesso di integrarmi perfettamente nel suo gruppo, ed il Prof. Marco Danelutto, sia per avermi concesso di svolgere presso il CERN il lavoro di tesi, che per la pazienza e la disponibilità avute durante questi mesi. Un ringraziamento particolare va al Dott. Giulio Eulisse, amico, collega e grande professionista, che mi ha sempre seguito ed aiutato durante il mio lavoro, fino alla fine della stesura di questa tesi. Un ringraziamento anche a tutte le persone con le quali ho lavorato e collaborato durante i miei due anni al CERN.

Ringrazio i miei amici di Pisa: Alessandro, Stefano di Larino, Stefano di Pisa, Carla, mio cugino Alessandro, Antonio, Paola, Manuela, Riccardo, Laura e Margherita.

Ringrazio i miei amici di Ginevra: Melò, mia cugina Francesca, Giovanni, Alessandro "le grand", Dora, Alessandro"le petit", Marco, Giulia, Marina, Andrea, Marcello, il mio amico "virtuale" Yann, Maria, Federica, Paolo, Gianluca ed infine la Cordeliere, luogo in cui abbiamo passato tutti insieme tanti bei momenti.

Bibliografia

[1] *CERN reference page.* http://www.cern.ch/.

[2] *Grid Cern wiki page.* https://twiki.cern.ch/twiki/bin/view/CMS/WorkBookStartingGrid.

[3] *Montecarlo event generator guide.* https://twiki.cern.ch/twiki/bin/view/CMS/WorkBookGenIn

[4] *The Concurrent Version System (CVS).* http://www.nongnu.org/cvs/.

[5] *SCRAM reference page.* http://cmsdoc.cern.ch/Releases/SCRAM/doc/scramhomepage.html.

[6] *NICOS reference page.* http://www.usatlas.bnl.gov/computing/software/nicos/.

[7] *COBRA framework reference page.* http://cobra.web.cern.ch/cobra/.

[8] *ORCA framework reference page.* http://cmsdoc.cern.ch/orca/.

[9] V. Innocente. *CMS Software Architecture: Software framework, services, and persistency in high level trigger reconstruction and analysis.* CMS/IN 1999-034.

[10] *GEANT reference page.* http://wwwasd.web.cern.ch/wwwasd/geant/.

[11] *OSCAR reference page.* http://cmsdoc.cern.ch/oscar/.

[12] *CMSSW framework reference page.* http://cmsdoc.cern.ch/cms/cpt/Software/html/General/.

[13] *The CMS High Level Trigger System.* IEEE transactions on nuclear science, Vol.55, No.1, February 2008.

[14] CMS Collaboration. *Computing Technical Proposal.* CERN/LHCC 1996-45 (1996).

[15] *LCG Project Application home page.* http://lcgapp.cern.ch/.

[16] CMS Collaboration. *The TriDAS Project Technical Design Report, Volume 1: The Trigger Systems.* CERN/LHCC 2000-38 (2000). CMS TDR 6.1.

[17] CMS Collaboration. *The TriDAS Project Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger.* CERN/LHCC 2002-26 (2002). CMS TDR 6.2.

[18] *ROOT reference page.* http://root.cern.ch.

[19] *XDAQ reference page.* http://xdaqwiki.cern.ch/index.php/Main_Page.

[20] *IGUANA reference page.* http://iguana.web.cern.ch/iguana.

[21] S. Muzaffar I. Osborne L. Taylor G. Alverson, G. Eulisse and L. A. Tuura. *IGUANA Architecture, Framework and Toolkit for Interactive Graphics.* ECONF C0303241 MOLT008 (2003). http://arxiv.org/abs/cs.se/0306042.

[22] *Design pattern produttore/consumatore.* http://zone.ni.com/devzone/cda/tut/p/id/3023.

[23] *Publish-subscribe pattern wikipedia page.* http://en.wikipedia.org/wiki/Publish/subscribe.

[24] *Rond robin scheduling wikipedia reference page.* http://en.wikipedia.org/wiki/Round-robin_scheduling.

[25] *Proxy server wikipedia reference page.* http://it.wikipedia.org/wiki/Proxy.

[26] *CMS Data Management Webtools.* https://twiki.cern.ch/twiki/bin/view/CMS/DMWebtools.

[27] *Javascript wikipedia reference page.* http://it.wikipedia.org/wiki/JavaScript.

[28] G. Eulisse. *Interactive Web-based Analysis Clients using AJAX.* CHEP 2006.

[29] *Multiple dispatch wikipedia page*. http://en.wikipedia.org/wiki/Multiple_dispatch.

[30] *SEAL library reference page*. http://seal.web.cern.ch/seal/.

[31] R.Watson. *Unix domain sockets versus internet sockets*. http://lists.freebsd.org/pipermail/freebsd-performance/2005-February/001143.html.