

python nell'esperimento GLAST

L. Baldini, J. Bregeon, C. Sgrò, R. Claus (ed il gruppo GLAST-Online)

Introduzione



- GLAST = Gamma-ray Large Area Space Telescope
 - Esperimento di (astro)Fisica su satellite
 - Progettato per la rivelazione di raggi gamma da sorgenti celesti
 - Data di lancio prevista: gennaio-febbraio 2008 (ci siamo quasi)
 - Durata della missione: 5-10 anni
- GLAST usa python!
 - Software online (acquisizione dati e verifica delle funzionalità)
 - Non solo...

Sommario

- Parte I (cosa?): che cosa è GLAST
 - Numeri interessanti e ordini di grandezza
 - Modello e fasi dello sviluppo
- PARTE II (perché?): perché Python
 - Perché: sia in senso di causa *formale* che di causa *finale*
 - Le regole del gioco: semplicità di utilizzo, stabilità, tracciabilità
 - Lo scopo del gioco: acquisire dati
- PARTE III (come?): dettagli (più) tecnici
 - Esempi concreti di utilizzo
 - Pacchetti aggiuntivi utilizzati, etc.

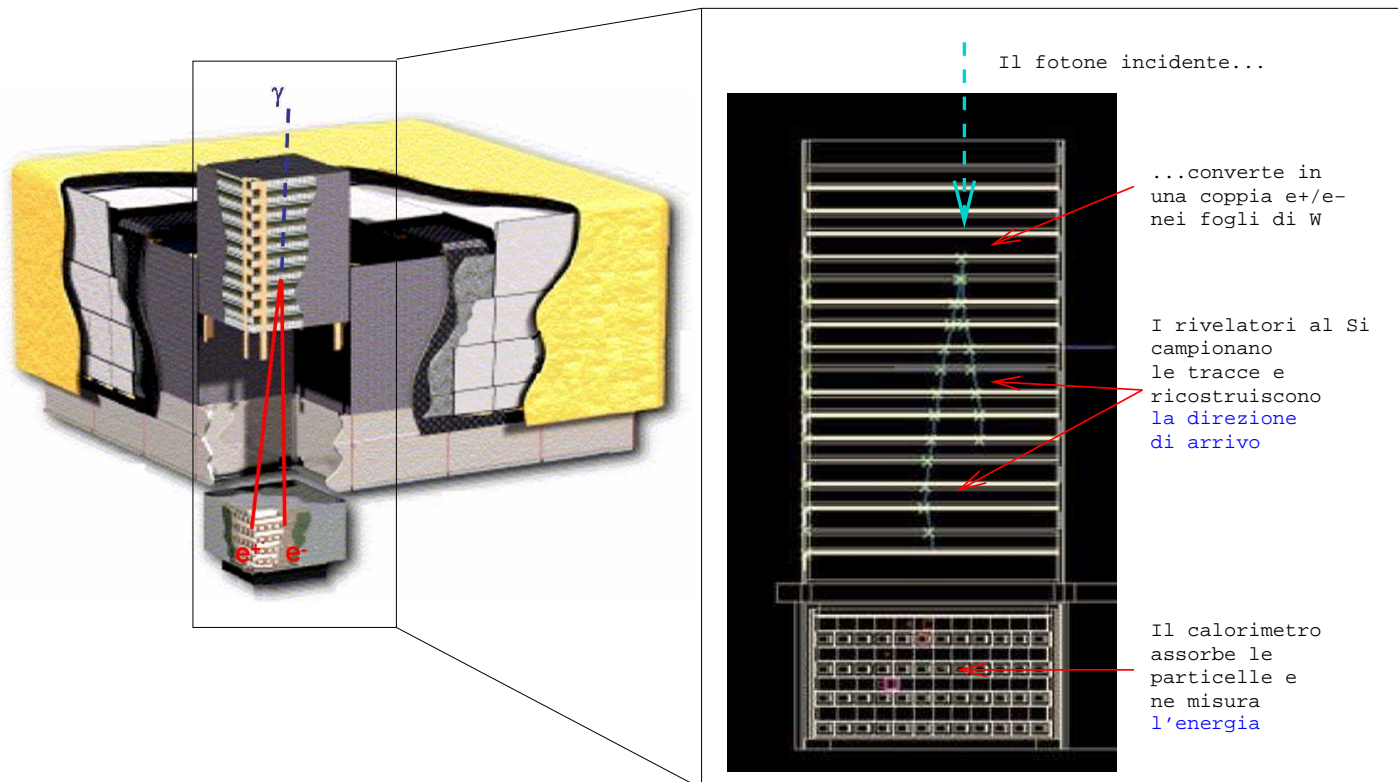
GLAST: la storia

- Dalla progettazione al lancio
 - Primo articolo scientifico: Space Science Rev. 75: 109-125, 1996
 - Fase di R&D (inclusi test su fascio ed un volo su pallone): 1999-2004
 - Costruzione (un intero sottosistema costruito in Italia) ed integrazione: 2004-2006
 - Integrazione con lo spacecraft: 2006-2007

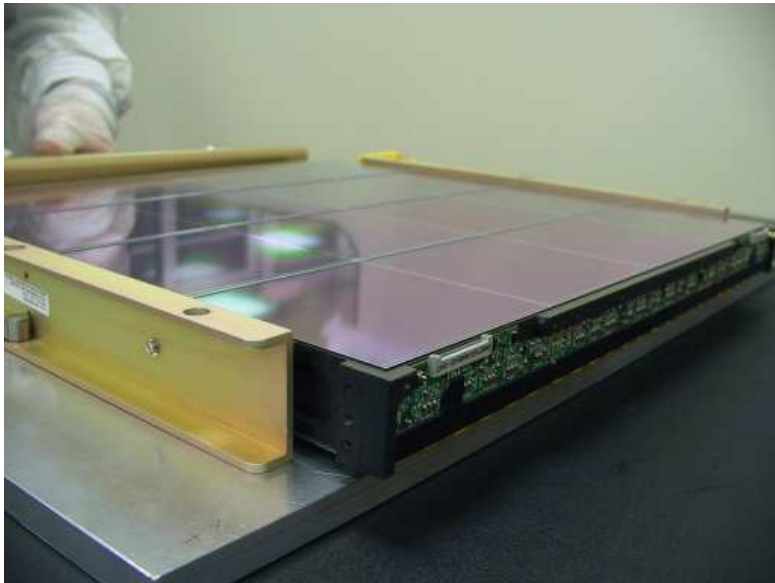


Come funziona?

- L'obiettivo è misurare direzione, tempo di arrivo ed energia dei raggi γ incidenti
 - Tre sottosistemi: Tracciatore, Calorimetro, Schermo di anticoincidenza
- Il nostro lavoro è rivelare particelle: nel seguito parleremo di acquisizione dati (DAQ), *monitoring* ed analisi
 - evento: risultato di una *lettura* completa del rivelatore
 - *run*: insieme di eventi acquisiti in una stessa configurazione o durante uno stesso test



Numerologia



- Il Large Area Telescope
 - Dimensioni: $\sim 2 \text{ m} \times 2 \text{ m} \times 1.5 \text{ m}$
 - Massa: ~ 3 tonnellate
 - Consumo: 650 W
 - ~ 300 collaboratori in svariati paesi del mondo
- Il tracciatore
 - Assemblato completamente in Italia
 - ~ 1 milione di canali di elettronica indipendenti
 - ~ 10000 rivelatori al silicio
 - ~ 15000 chip di elettronica di lettura (ASIC) dedicati
 - ~ 70000 registri indipendenti (e tutti diversi...) per la gestione della configurazione.

Fasi dello sviluppo

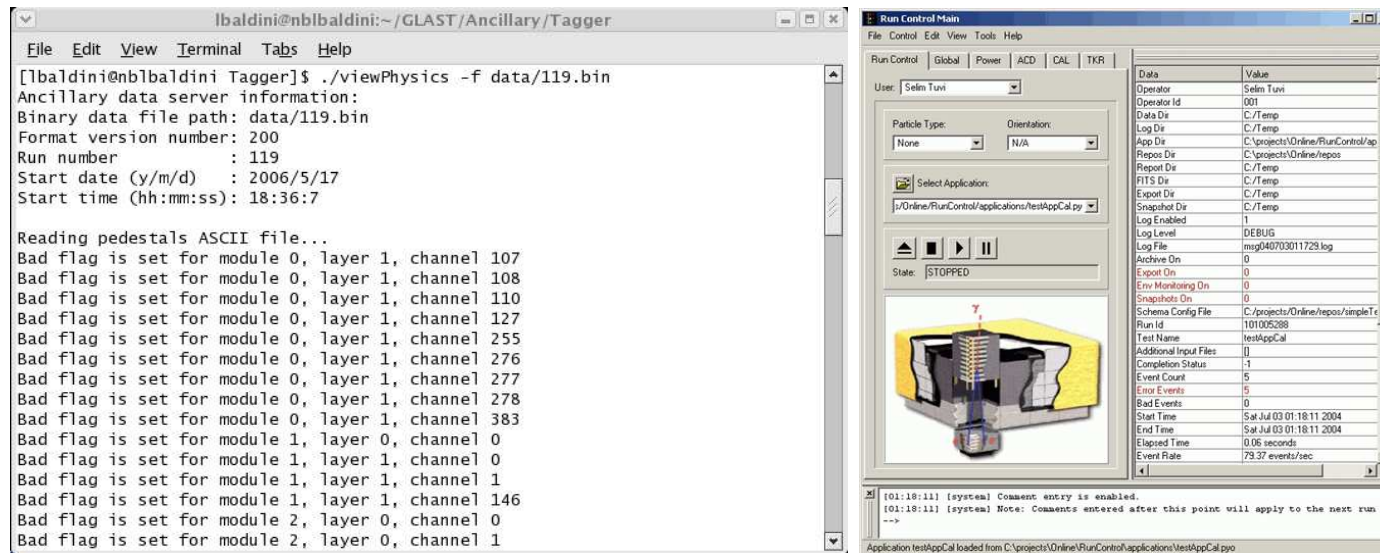
- Produzione, assemblaggio e test dei singoli sottosistemi in parallelo in diversi paesi
 - Italia, Giappone, U.S.A. (tracciatore)
 - Francia, Svezia, U.S.A. (calorimetro)
 - U.S.A. (schermo di anticoincidenza)
- Integrazione e test del rivelatore
 - U.S.A. (SLAC, CA)
 - Idea di base: i test definiti al livello dei singoli sottosistemi sono ripetuti dopo l'integrazione
- Integrazione sullo *spacecraft*
 - U.S.A. (General Dynamics, Phoenix AZ)
- Il modello di sviluppo del software di acquisizione dati/test deve tener conto di tutto questo
 - Necessità di un *framework* comune: robusto, semplice, mantenibile



Requisiti

- Semplicità di sviluppo
 - Numerosi sviluppatori distanti tra loro, ognuno con le proprie idiosincrasie
 - Necessità di convergere, attraverso le varie fasi della costruzione, ad un *tool* comune
- Completa tracciabilità
 - Possibilità di risalire in qualsiasi momento, ai dettagli della configurazione in cui un test specifico è stato eseguito
 - Inclusa la versione del software !
 - Generazione di *test report* per ogni test effettuato
- Robutezza/semplifictà di utilizzo
 - Si assume che gli operatori non sappiano cosa stanno facendo, ma siano in grado di seguire istruzioni
 - Criteri automatici e chiari di `PASS/FAIL` per ciascun test

DAQ: prima e dopo python (slide semiseria)



Prima

- Command line (con una qualche forma di GUI in Tk/Tcl nei casi più fortunati)
- Un unico file in fortran o C
- Praticamente impossibile da mantenere e modificare

Dopo

- Scritto in maniera modulare e mantenibile
- Interfaccia utente (quasi) *idiot proof*
- Possibilità di *display* e monitor dei dati in tempo *reale*

Tool utilizzati

● Core

- Quasi interamente scritto in python, utilizza molte delle funzionalità della libreria standard
- Alcune funzionalità specifiche implementate sotto forma di classi C++ (per ragioni di velocità) esposte al resto del framework come librerie compilate (usando SIP/SWIG)

● Interfaccia utente

- Interamente realizzata con Qt/pyQt (supporto per applicazioni *multithread*, segnali e *slot*, internazionalizzazione, *designer* completo, ottima documentazione)

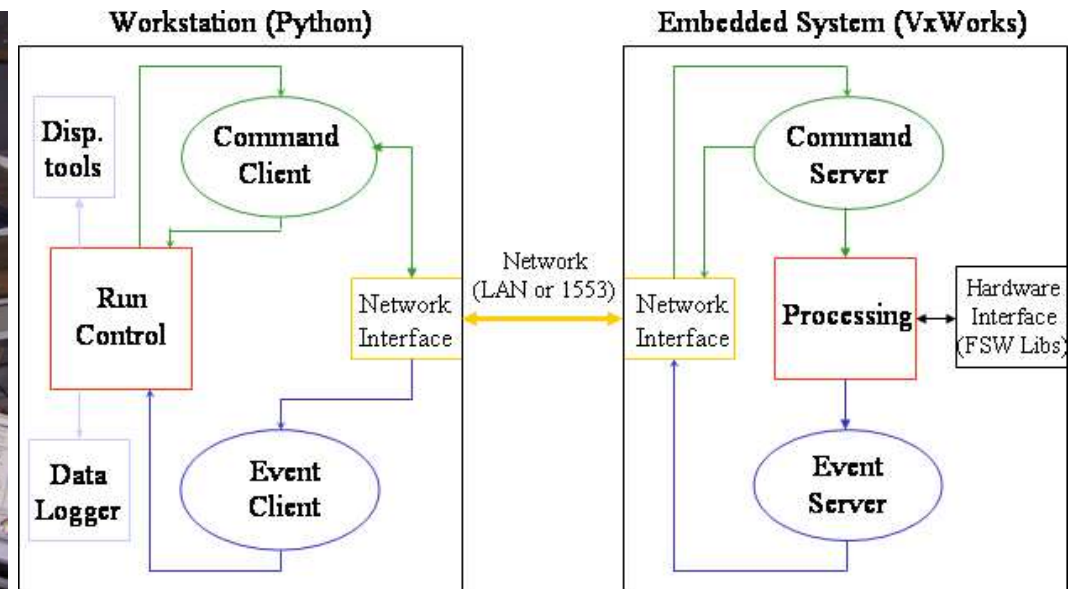
● Analis/visualizzazione dati

- Implementata principalmente tramite Hippodraw (pacchetto sviluppato in C++, ha un *wrapper* in python, utilizza Qt ed ha un widget specifico per Qt/pyQt)
 - <http://www.slac.stanford.edu/grp/ek/hippodraw>
- pyROOT (occasionalmente)
 - <http://root.cern.ch>

● Pacchetti aggiuntivi

- numpy (essenzialmente per gestire array)
- pyFITS (per gestire file FITS, formato particolarmente amato dagli astrofisici; se avete tempo e coraggio date un'occhiata...)

Architettura del DAQ: le due facce



- Gestione a basso livello dell'hardware
 - Routine compilate in C
 - Sistema Operativo in tempo reale : VxWorks
 - Eseguite fisicamente su un processore VME *embedded*
- Framework di sviluppo e test
 - Completamente scritto in python
 - Sviluppato in modo del tutto indipendente
 - Eseguito su una normale *workstation*

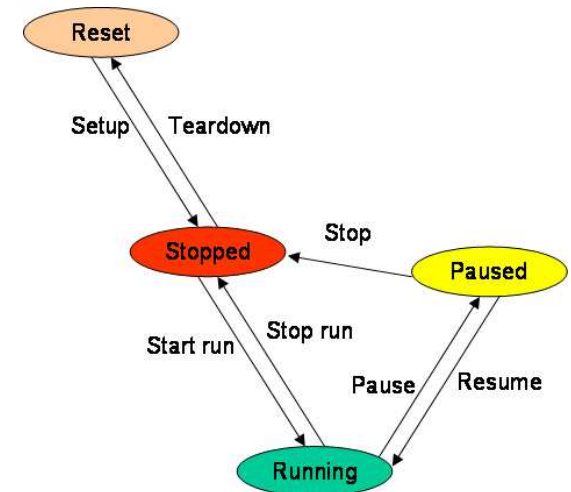
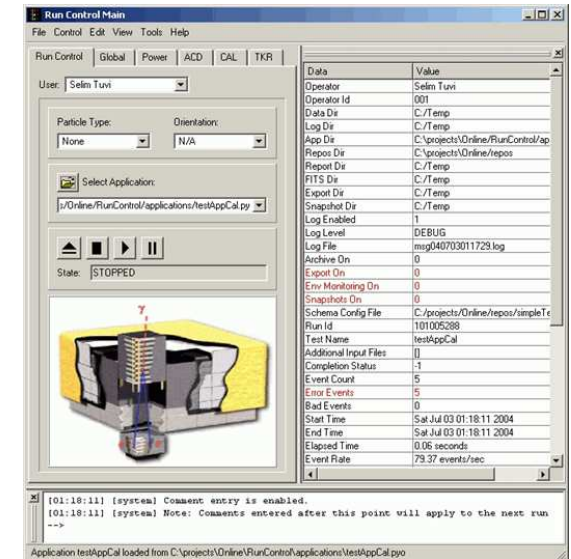
Run Control

Run control

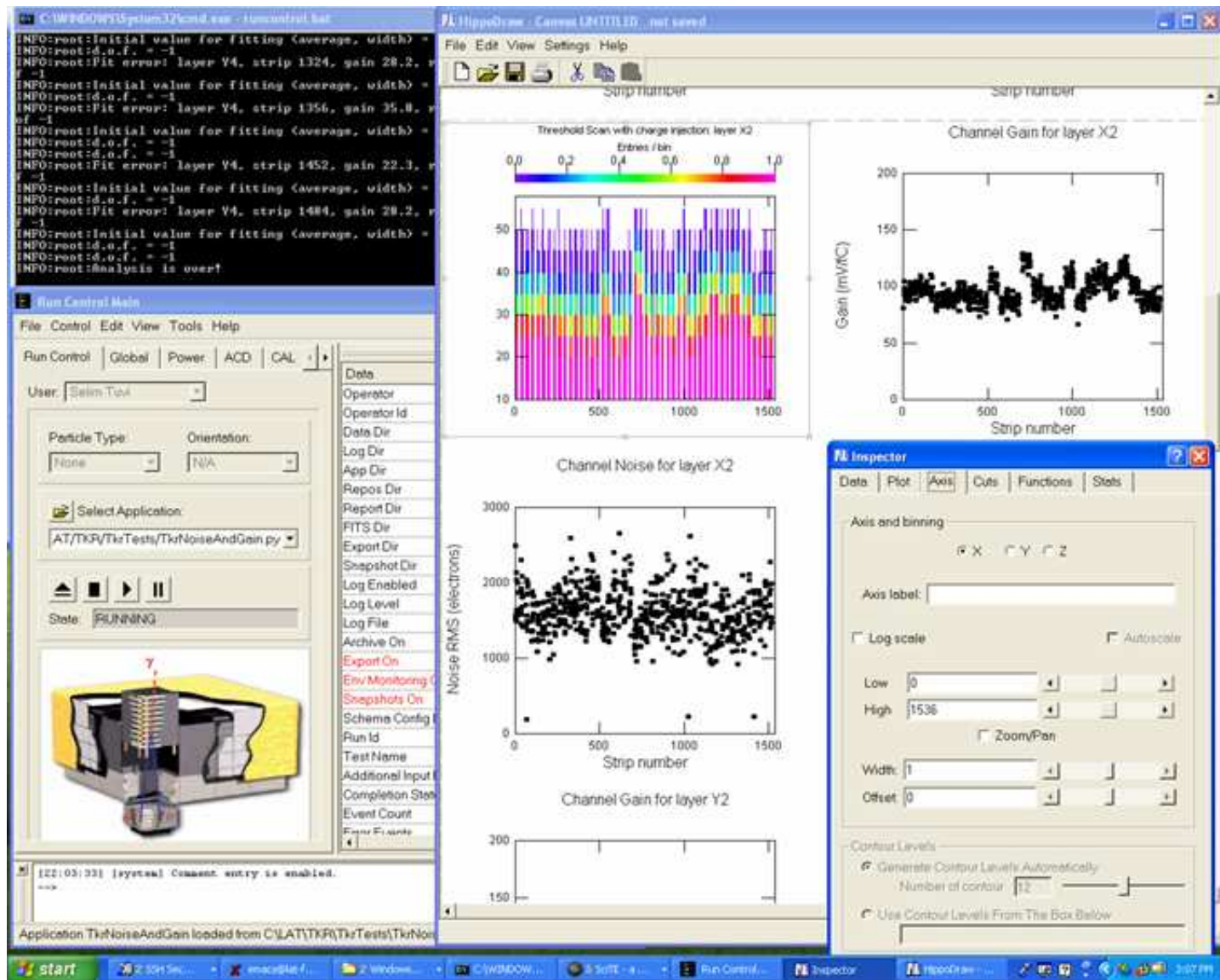
- Ambiente di scripting basato su una macchina a stati finiti
- Monitor dello stato dello strumento
- Archiviazione e distribuzione dei dati
- Shell di python per accesso *diretto* all'hardware (utile per *debug*)

Applicazioni di test

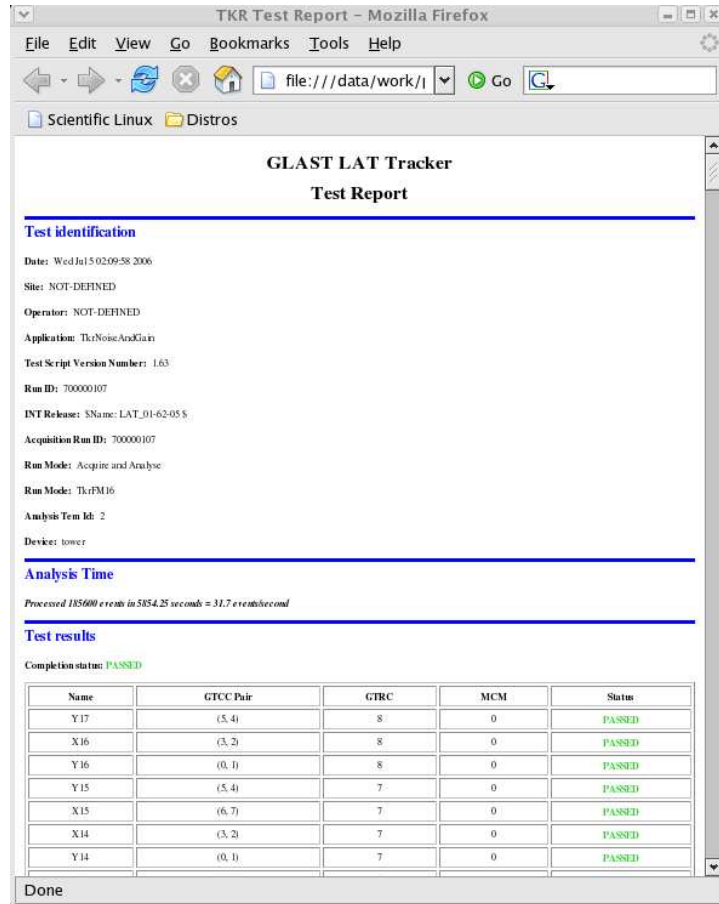
- Ereditano da una classe base (*userApplication*)
- Implementano le operazioni da eseguire in corrispondenza di ciascuna transizione di stato
- Specificano i dettagli del processamento dei dati
- Implementano (se necessario) la visualizzazione online dei prodotti del test nell'ambito del *framework* comune



Un esempio di applicazione



Test report



TKR Test Report - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

file:///data/work/1

Scientific Linux Distros

GLAST LAT Tracker Test Report

Test identification

Date: Wed Jul 5 02:09:58 2006
Site: NOT-DEFINED
Operator: NOT-DEFINED
Application: TkrNoiseAndGain
Test Script Version Number: 1.63
Run ID: 700000107
INT Release: SName: LAT_01-62-05 S
Acquisition Run ID: 700000107
Run Mode: Acquire and Analyse
Run Mode: TkrFM16
Analysis Test Id: 2
Device: tower

Analysis Time

Processed 185600 events in 5854.25 seconds = 31.7 events/second

Test results

Completion status: PASSED

Name	GTCC Pair	GTRC	MCM	Status
Y17	(5, 4)	8	0	PASSED
X16	(3, 2)	8	0	PASSED
Y16	(0, 1)	8	0	PASSED
Y15	(5, 4)	7	0	PASSED
X15	(6, 7)	7	0	PASSED
X14	(3, 2)	7	0	PASSED
Y14	(0, 1)	7	0	PASSED

Done

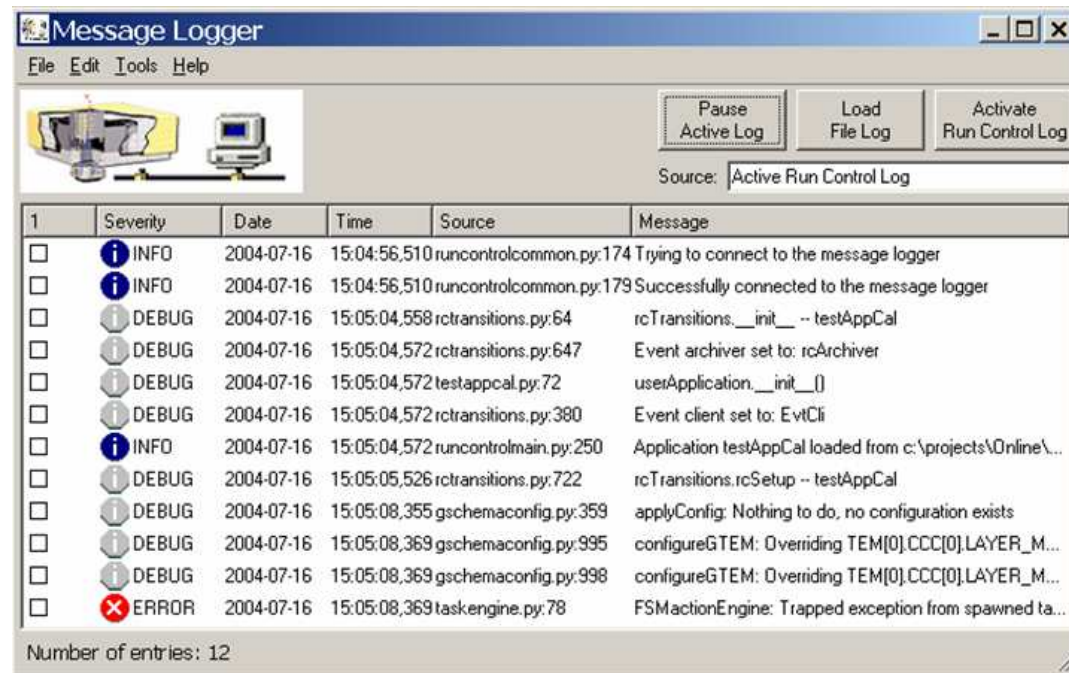
- Sommari dettagliati (in formato html) prodotti alla fine di ogni run
- Permettono di risalire, in qualsiasi momento, alle condizioni esatte in cui ogni test è stato eseguito e all'esito del test stesso
 - Contestualmente tutti i file rilevanti (file di configurazione xml, messaggi del logger, *snapshot* dell'hardware) sono archiviati insieme al *test report*
 - Contengono testo, tabelle, grafici e *link* a materiale di vario genere
- Automaticamente archiviati in un *database* dedicato
- python fornisce un insieme di funzioni che facilitano la manipolazione delle stringhe

Configurazione hardware (`xml.dom`, `xml.sax`)

- Definizione della configurazione
 - Implementata interamente in formato xml
 - Run Control si occupa del parsing e dell'applicazione della configurazione
 - Configurazione diverse possono essere selezionate ed applicate *al volo*
- Tracciabilità della configurazione
 - Due *snapshot* completi dello stato del sistema (all'inizio ed alla fine di ogni test) generati automaticamente
 - File xml corrispondenti archiviati in maniera permanente

```
<GTEM>
  <cal_trgseq>0x2b</cal_trgseq>
  <GTIC>
    <cal_in_mask>0x0</cal_in_mask>
    <cal_biasdac>0x5800</cal_biasdac>
    <cal_lrs_mask>0x0</cal_lrs_mask>
  </GTIC>
  <GCCC>
    <configuration>0xAE</configuration>
    <GCRC>
      <dac>0x0</dac>
      <delay_3>0x85</delay_3>
      <GCFE>
        <config_1>0x7</config_1>
      </GCFE>
    </GCRC>
  </GCCC>
</GTEM>
```

Notifica dei messaggi (logging)



- Permette di tracciare completamente le singole operazioni di test
- Livelli di *screening* diversi a seconda dell'output (terminale, file)
- GUI custom sviluppata per la visualizzazione interattiva

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info('hello world!')
```

Browsing dei registri (threading)

- Applicazione indipendente *lanciabile* dalla GUI principale
- Eseguita fisicamente su un thread indipendente
- In *polling* sui registri del hardware

```
import threading
```

```
from time import sleep
```

```
new_thread = \
```

```
    threading.Thread(None, loop, 'loop')
```

```
self.new_thread.start()
```

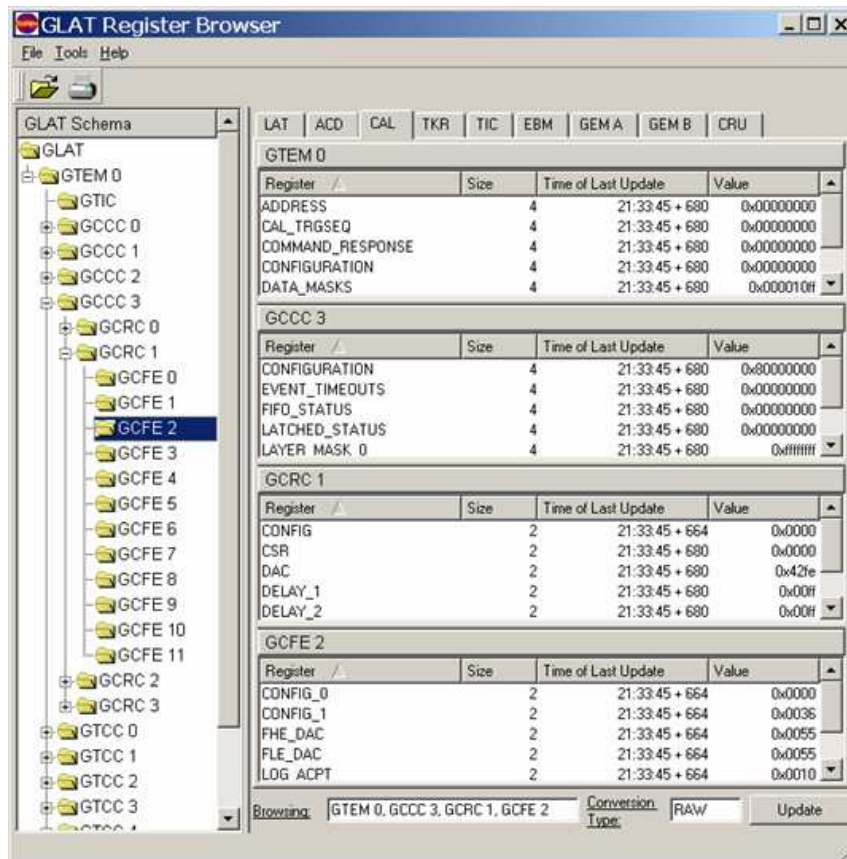
```
def loop():
```

```
    read_registers():
```

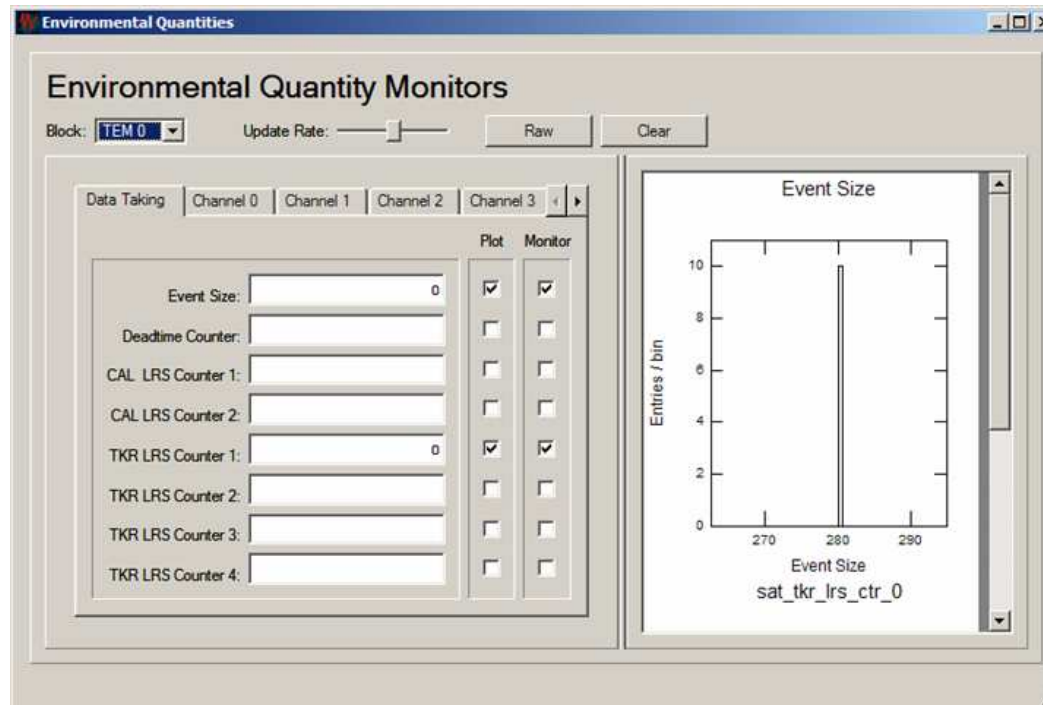
```
    sleep(1)
```

```
def read_registers():
```

```
    ...
```

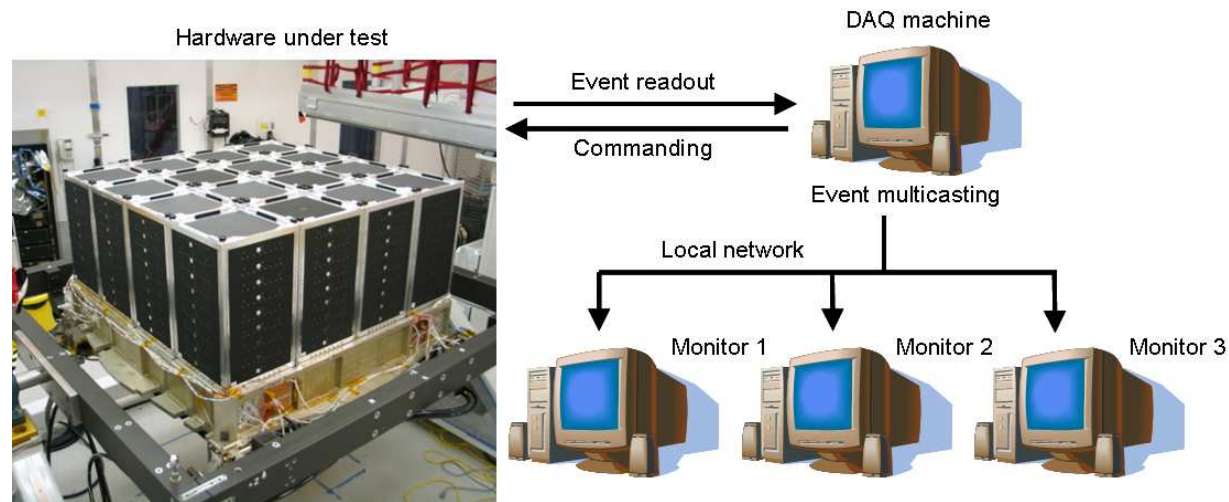


Monitor ambientale (threading)



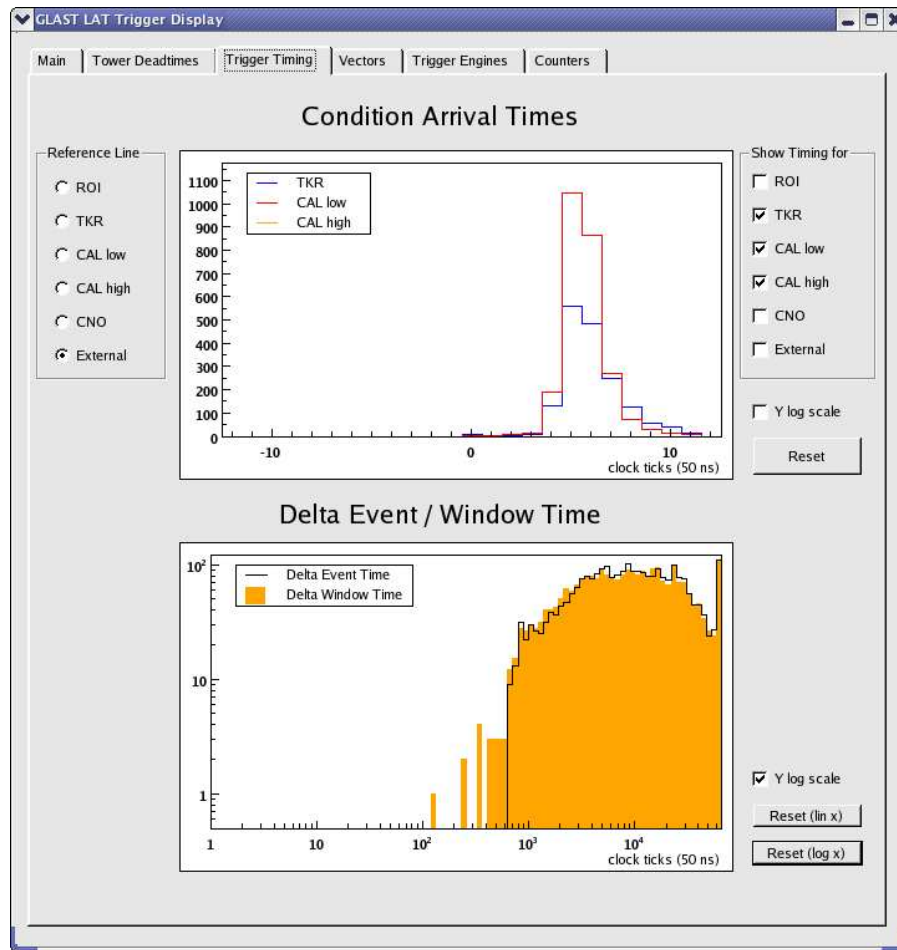
- Esigenze simili al *browsing* dei registri (necessità di gestire *thread* multipli con carico di CPU modesto e senza vincoli temporali stringenti)
- Permette di monitorare in tempo reale tutte le tensioni, correnti e temperature interessanti
- Supporto grafico per la visualizzazione (possibilità di creare al volo istogrammi e *strip chart*)

Monitor dei dati: *multicast* (socket)



- Run Control è in grado di *forwardare* i dati su un *socket* UDP da cui applicazioni indipendenti possono prelevarli
- Modello *producer/consumer* con le due applicazioni sulla stessa macchina o macchine diverse su una rete locale
 - Possibilità di avere una macchina dedicata *esclusivamente* all'acquisizione dei dati (con un *overhead* minimo causato dal *multicasting*)...
 - ... e una (o più di una, magari configurate diversamente) macchina per il monitor dei dati (lavoro intenso in termini di CPU)
- DAQ e monitor possono essere completamente disaccoppiati per la massima flessibilità

Monitor del *trigger*



- Il *trigger* è un elemento fondamentale di ogni esperimento di fisica
 - Sostanzialmente ci dice in *quale* sottosistema è successo qualcosa
- Il monitor del *trigger* è un esempio di applicazione *consumer* per i dati prodotti da Run Control (*producer*)
 - In questo caso le applicazioni vengono eseguite sulla stessa macchina
 - Il monitor del *trigger* si può lanciare dalla GUI principale di Run Control

Documentazione del codice (pydoc)

Fast monitor: pXmlOutputList.pCUSTOMXmlRep Class Reference - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

file:///data/work/ISOC/dataMonitoring/FastMon/python/doc/html/classpXmlOutputList_1_1pCUSTOMXmlRep

Scientific Linux Distro

pXmlOutputList.pCUSTOMXmlRep

pXmlOutputList.pCUSTOMXmlRep Class Reference

Class describing the representation of a custom plot. [More...](#)

Inheritance diagram for pXmlOutputList.pCUSTOMXmlRep:

```
graph BT
    pXmlOutputList_pCUSTOMXmlRep[pXmlOutputList.pCUSTOMXmlRep] --> pXmlOutputList_pPlotXmlRep[pXmlOutputList.pPlotXmlRep]
    pXmlOutputList_pPlotXmlRep --> pXmlElement_pXmlElement[pXmlElement.pXmlElement]
```

[List of all members.](#)

Public Member Functions

def	__init__
	Constructor.
def	getRootObject
	Return the custom ROOT histogram.

- Indispensabile in una collaborazione di dimensioni medio-grandi
- `pydoc` è stata la scelta iniziale
- Doxygen (<http://www.stack.nl/~dimitri/doxygen>) scelto dopo le prime fasi di implementazione
 - All'inizio richiedeva un *pre-processing* dei sorgenti
 - Adesso doxygen supporta python in modo nativo
 - Utile per i progetti misti python/C++

Interazione con l'OS (`os`, `commands`)

- Il modulo `OS` fornisce un'interfaccia largamente *cross platform* alle funzionalità del sistema operativo
 - Grazie ad `OS.path` siamo liberi dalla schiavitù dello *slash*!
- Nel nostro caso l'intero sistema è migrato in modo *indolore* da Windows a Linux nel bel mezzo dello sviluppo!
 - Grazie a python e non solo...
- Nei casi più disperati rimane sempre `os.system`
- Lavorando sotto Unix python fornisce anche il modulo `commands`

```
import os
```

```
os.listdir('.')
```

```
os.makedirs('~/.test/hello')
```

```
os.system('cd ~; make')
```

```
...
```

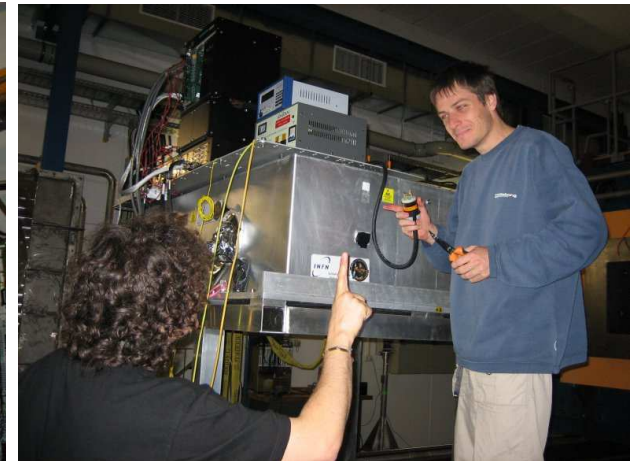
```
import commands
```

```
print command.getoutput('svn status')
```

Un'applicazione specifica: il *beam test*

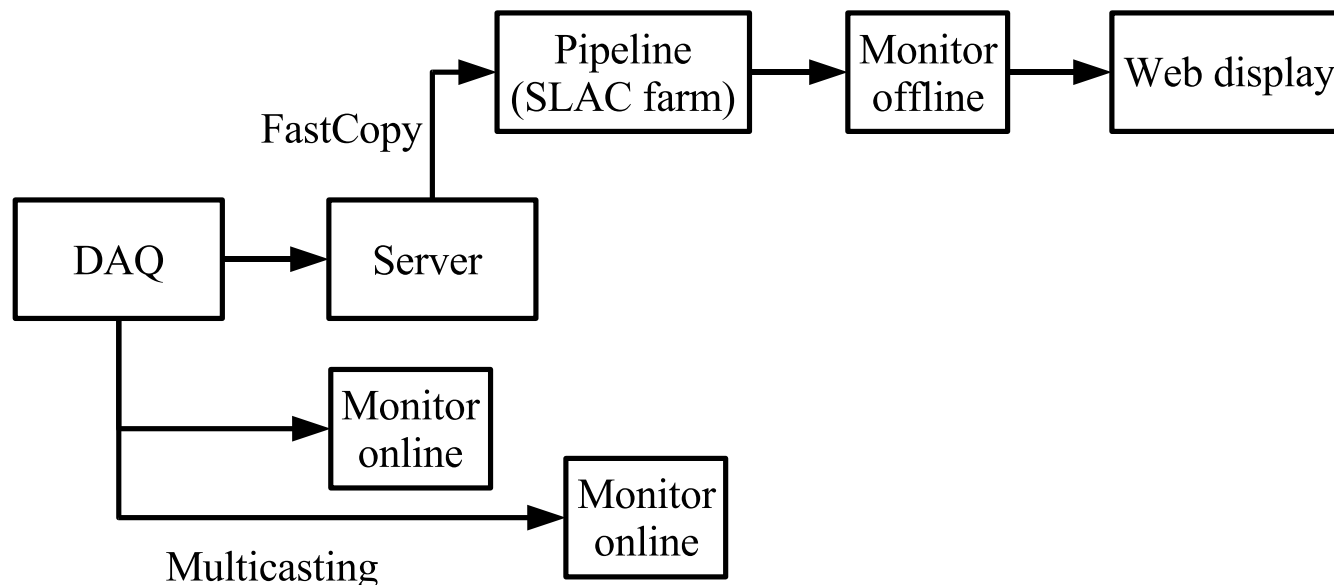
- Letteralmente: test su fascio (di particelle prodotto da un acceleratore)
 - Slot di tempo ben precisi (e non rinegoziabili) assegnati
 - Il tempo di fascio è prezioso e non recuperabile: non è permesso sbagliare
 - La capacità di adattarsi all'ambiente è un punto cruciale
- Il software di acquisizione e monitor dei dati deve essere abbastanza flessibile da adattarsi alle esigenze (difficilmente prevedibili a priori)
- Due campagne estensive di test per la calibrazione dello strumento: CERN (Ginevra) nell'estate 2006 e GSI (Darmstadt) nell'autunno 2006
 - Necessario per caratterizzare le prestazioni dello strumento con sorgenti note...
 - ... e per validare la simulazione dello strumento stesso
 - Un gruppo di ~ 40 persone impegnato per un mese e mezzo (tempo *biblico* per un *beam test*, modesto per il ciclo di utilizzo di un pacchetto *software*)

Cartoline dal *beam test*



Beam test: il flusso dei dati

- Due percorsi paralleli per il monitor dei dati
 - Archiviazione, *fast-copy* a SLAC (California), processamento completo con una farm dedicata, esecuzione del codice di monitor *offline*. *Round-trip*: qualche ora
 - Monitor *online*, sfruttando la capacità di multicasting del DAQ. In tempo reale
- Davide vs. Golia
 - Catena *offline* completa: diverse centinaia di migliaia di righe di C++
 - Poche migliaia di righe di python
 - Grandezze fondamentali ricostruite con una differenza contenuta in termini di precisione (tipicamente un fattore 2)



Beam test: il monitor online

● Requisiti

- Configurabilità (deve supportare l'aggiunta di nuove variabili e/o grafici *on the fly*)
- Velocità (frequenza degli eventi da processare estremamente variabile, da 0.1 Hz a ~ 1 kHz)
- Semplicità di utilizzo (usato da operatori, non sviluppatori)

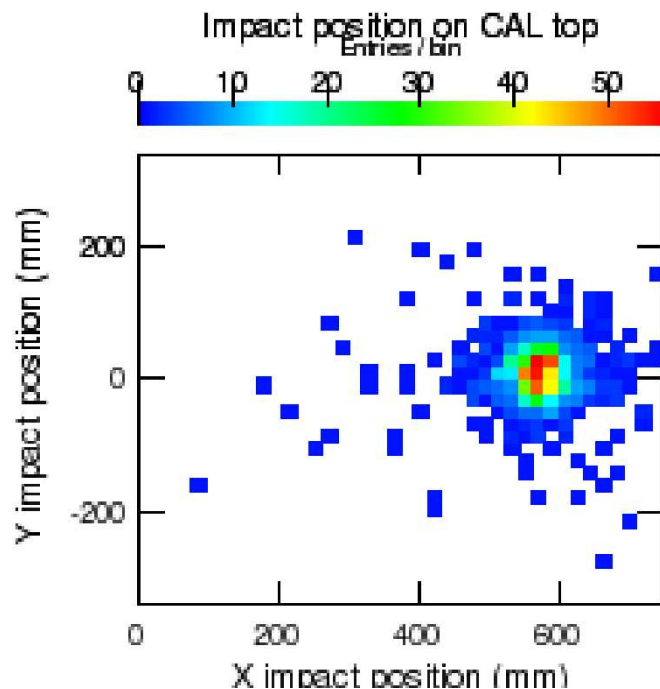
● Livello di *processing* non trascurabile

- *Parsing* dei dati
- Conversione dallo spazio dell'elettronica allo spazio fisico
- Applicazione delle costanti di calibrazione
- Ricostruzione degli eventi

● Architettura

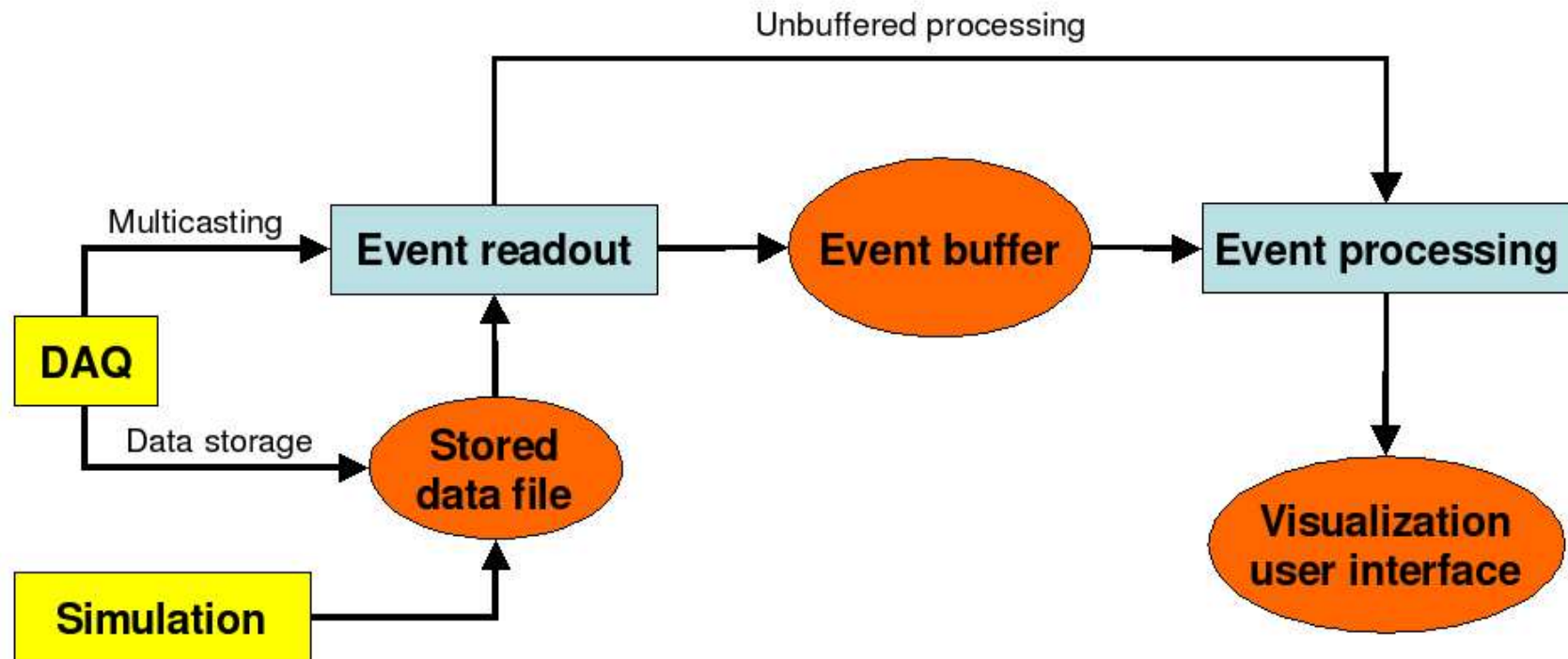
- Un file di configurazione unico contenente liste e dizionari python (`eval` ed `exec` fanno il resto)
- Due thread concorrenti che accedono asincronamente ad un buffer di eventi comune
- GUI realizzata con PyQt: tab multipli per grafici e tabelle, tooltips...

Monitor *online*: configurazione



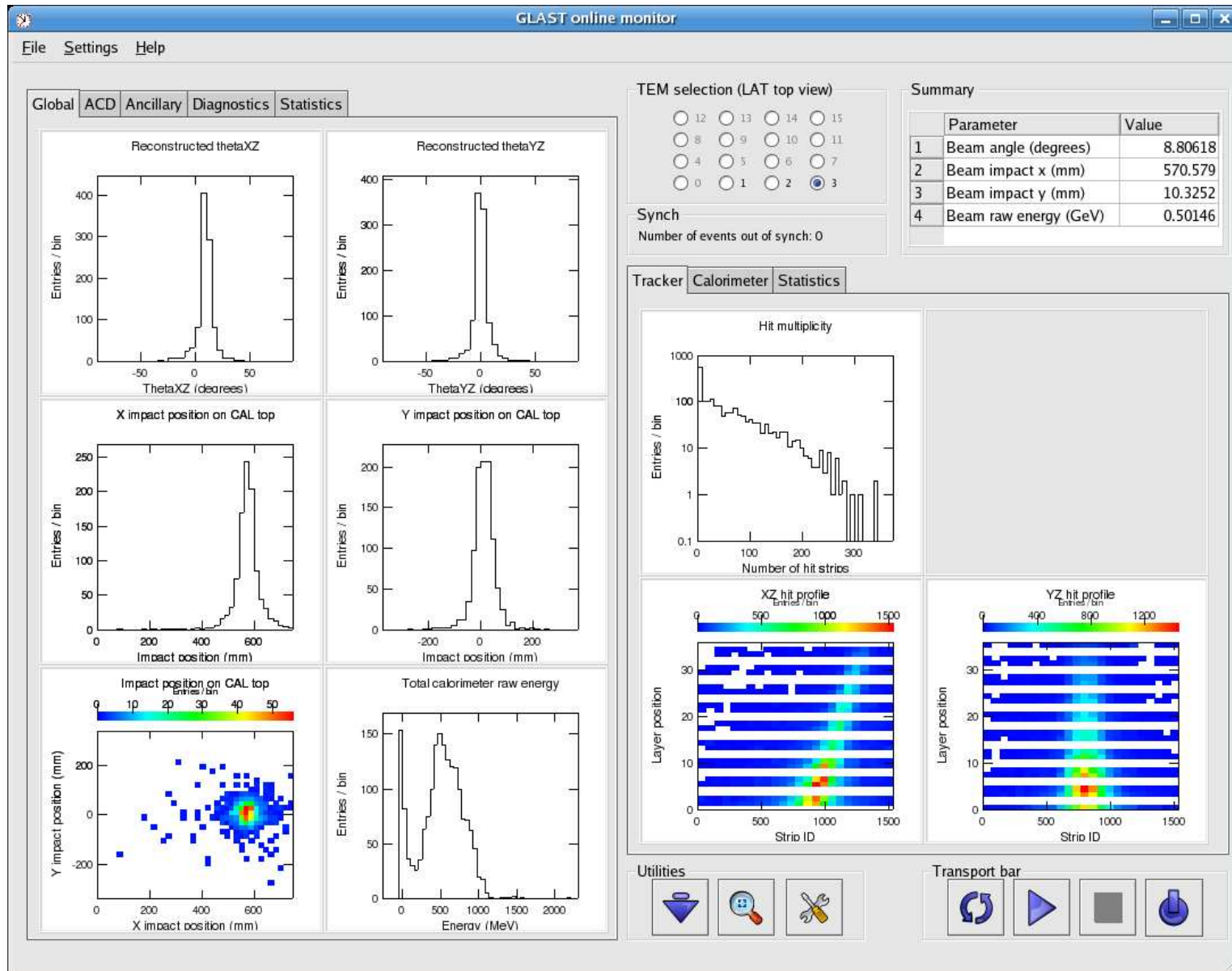
```
PLOTS_LIST =  
{  
  "evt_impact_point" : {  
    "Type"      : "ColorPlot",  
    "NTuple"    : "Main",  
    "Settings" : '[ "NODE_evt_impact_point_x", '  
                  "NODE_evt_impact_point_y"]',  
                  (-0, 800), 40, '  
                  (-400, 400), 40, '  
                  (None, None), '  
                  "Impact position on CAL top", '  
                  "X impact position (mm)", '  
                  "Y impact position (mm)"',  
    "ToolTip"   : "LAT XY coordinate"}  
}
```

Monitor *online*: buffering degli eventi



- Applicazione *multithread* (tipo *producer/consumer*)
- Buffer implementato come un oggetto di tipo `Queue`, modificato per adattarsi automaticamente al *rate* di dati in ingresso
- Risorse di sistema automaticamente bilanciate tra i due thread a seconda del rate istantaneo

Monitor *online*: GUI ... (thx PyQt)



I/O dei dati (`pickle`, `cPickle`)

- Permette di salvare su file oggetti (quasi) arbitrariamente complicati...
- ...incluse istanze di classi definite dal programmatore
- Risolve il problema dell'I/O dei dati, per lo meno quando non si deve comunicare con applicazioni esterne
- Piccolo “database” dei *run* del *beam test* implementato letteralmente in poche ore utilizzando `cPickle` e poco altro della libreria standard

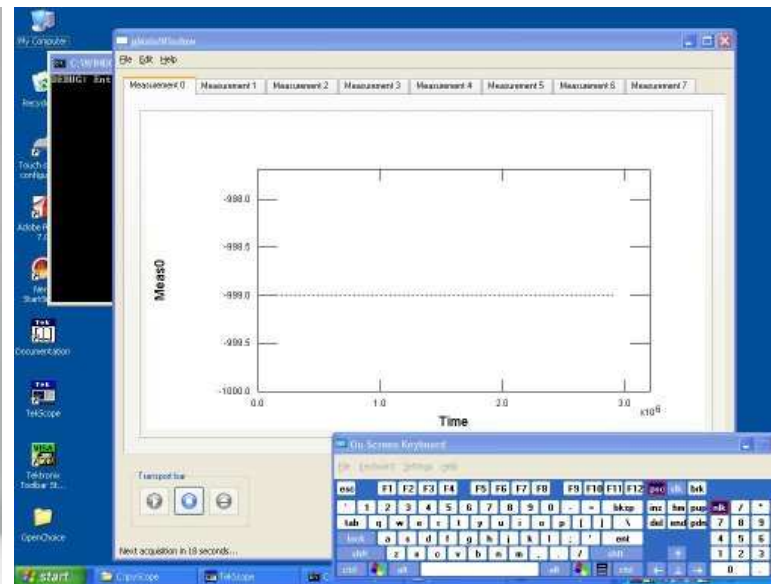
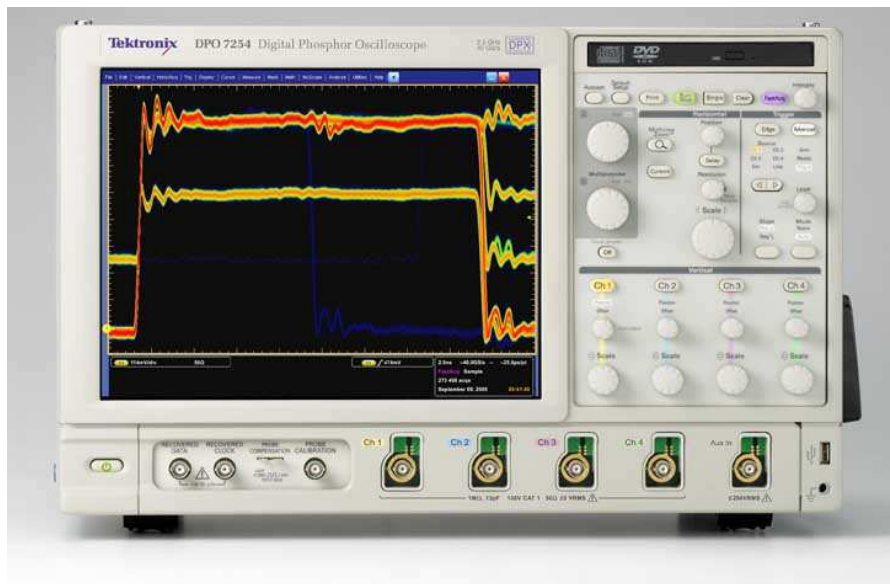
```
import cPickle

object = whatever_you_want()

cPickle.dump(object, file('test.pickle', 'w'))

...

object = cPickle.load(file('test.pickle', 'r'))
```



- VISA è uno standard per l'interfaccia con strumenti di misura
 - VISA = VXIplug&play System Alliance
 - VXI = VME eXtension for Instrumentation
 - VME = VERSAmodule Eurocard bus
 - ...
- Il modulo pyVisa permette di controllare strumenti attraverso svariati *bus*: GPIB, RS232, USB, RJ45...
 - <http://pyvisa.sourceforge.net>

python greatest hits

● Standard library

- atexit
- bisect
- math
- optparse
- random
- re

● Pacchetti esterni

- numpy (<http://numpy.scipy.org>)
- orange (<http://ailab.si/orange>)
- pyparallel (<http://pyserial.sourceforge.net>)

Conclusioni

- Facile da imparare
- Ha una libreria standard sconfinata
- Molti pacchetti hanno wrapper in python (Qt, ROOT, Hippodraw)
- Consente tempi di sviluppo brevi anche per applicazioni complesse
- Cross platform

⇒ GLAST usa python in modo intensivo

<http://glast.gsfc.nasa.gov/>

